SEQUENT

DYNIX/ptx
C and Language Tools

1003-48544-00

THIS SHEET LIFTER
MUST BE PLACED ON
TOP OF CONTENTS

GRAPHIC PRODUCTS
Chicago, Illinois 60609

DYNIX/ptx®
C and Language Tools

C User's Manual
1003-48545-00

lint Reference Manual
1003-49927-00

ld User's Guide
1003-49928-00

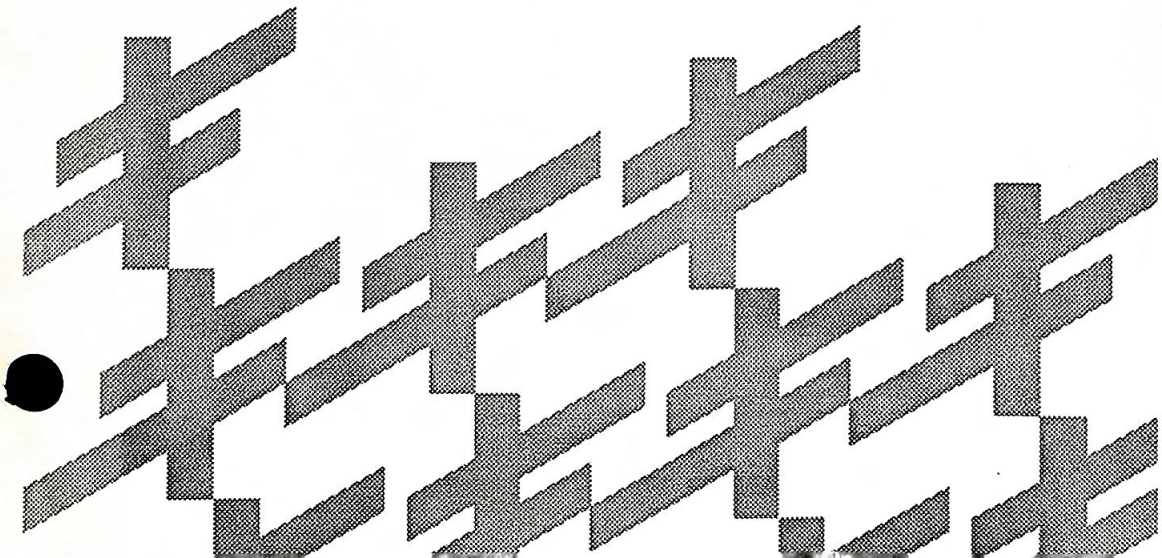
Assembly Language User's Manual
1003-48546-00

Pdbx User's Manual
1003-48547-00

C and Language Tools Man Pages
1003-49195-00

C and Language Tools
Version 1.8p Release Notes
1003-48550-00

Sequent Computer Systems, Inc.



Note to the Reader

In a typical development sequence, a programmer may write a program in C, check the correctness of the program using **lint**, compile and link the program using **cc**, execute the program, and, if the program does not execute as expected, debug the program with **pdbx** or **dbx**. The programmer may also use other tools, such as **make** and **awk**, to automate program development and execution. The manuals in this binder describe the following tools, which are commonly used to develop programs in the DYNIX/ptx environment:

- **cc** – C compiler
- **lint** – C syntax checker
- **ld** – linker
- **as** – assembler
- **pdbx, dbx** – debugger

In addition, the following tools are described in the man page section of this binder:

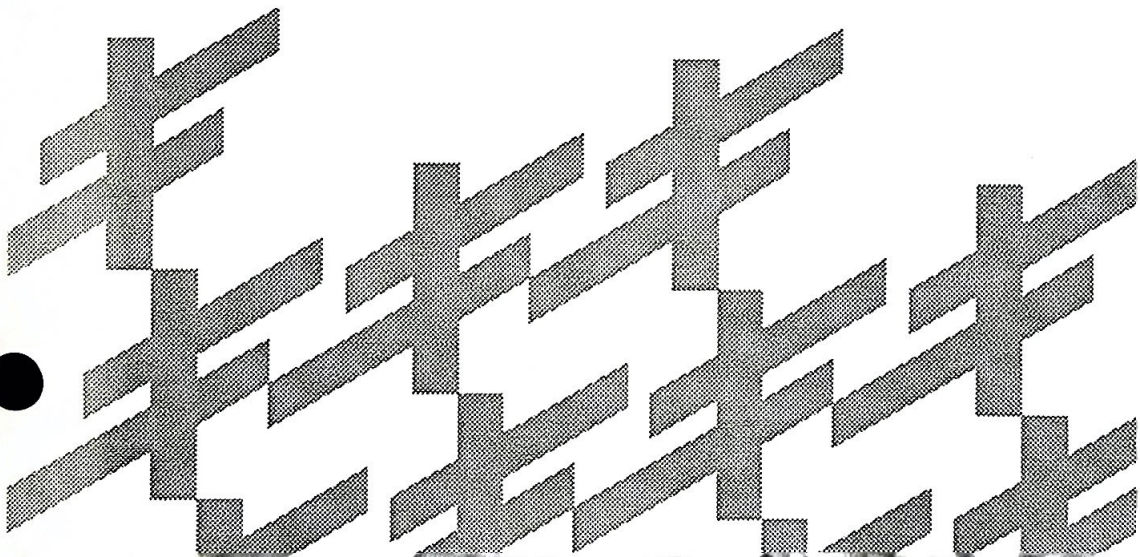
ar	**dis**
cb	**dump**
cflow	**list**
cpp	**lorder**
cprs	**nm**
ctrace	**size**
cxref	**strip**

For information about more general-purpose tools, including **make** and **awk**, refer to the *Programming Tools Guide*.

The documentation in this binder is written for experienced programmers; no attempt is made to teach a particular language or the basics of programming on a UNIX system. For introductory information on programming in the DYNIX/ptx environment, refer to the *Programming Guide*.

DYNIX/ptx®
C User's Manual

1003-48545-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

Symmetry is a trademark of Sequent Computer Systems, Inc.

UNIX is a registered trademark of AT&T.

VAX is a trademark of Digital Equipment Corporation.

Copyright 1988, 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America

Contents

About This Manual
Overview v
Assumptions about the Reader v
Organization v
Related Publications vi

Chapter 1. DYNIX/ptx C
1.1 Introduction 1-1
1.2 Supported Extensions 1-1
1.3 Porting Considerations 1-10
1.4 Running the Compiler 1-15

Appendix A. DYNIX/ptx C Language Calling Conventions
A.1 Calling Sequence A-1
A.2 Calling Conventions A-1
A.3 Argument List A-3

Appendix B. Assembly Language Code in C Programs
B.1 Assembly Language Directives B-1
B.2 Assembly Language Macros B-2

Figures

1-1 Valid and invalid uses of structure assignment 1-2
1-2 Example of the enum type specifier 1-4
1-3 Void type specifier 1-5
1-4 Example of a predefined name 1-5
1-5 Example of const referring to a single declaration 1-7
1-6 Example of const referring to an aggregate type 1-7
1-7 Test to determine type of assignment to unsigned short 1-11
1-8 Incorrect usage of asm statements in a C function 1-13

B-1 Example of an asm statement B-1
B-2 Example of an asm macro definition B-3
B-3 Using the asm macro in a C program B-4
B-4 Using patterns to specify different storage modes B-5

Tables

1-1 Using Modifiers with Pointers 1-8

B-1 Generated Assembly Code B-4

About This Manual

Overview

This manual describes DYNIX/ptx C, the Sequent C language for the DYNIX/ptx operating system. Hereafter, the DYNIX/ptx C compiler will be called simply the compiler. This manual describes how to invoke the compiler and explains its features.

Assumptions about the Reader

This manual assumes that you have a working knowledge of the C programming language; therefore, it does not teach C. The "Related Publications" section at the end of this preface contains a list of reference material that supplements this manual.

Organization

This manual is organized as follows: The **introduction** to the compiler is in Section 1.

Extensions that are part of the compiler are described in Section 2. These include features of the proposed ANSI standard as well as Sequent extensions to C.

Considerations for porting C programs to the DYNIX operating system are in Section 3.

Invoking the compiler is described in Section 4.

Calling conventions for the compiler are in Appendix A.

Assembly language code in C programs is described in Appendix B.

Runtime errors that can be returned by the routines in the C math library and a description of how to create your own routine for handling such errors are in Appendix C.

Notational Conventions

The following notational conventions are used in this manual:

bold	Words in bold are commands and compiler invocation options that you would type at your terminal exactly as shown.
italic	Italicized words indicate variables in a program, procedure names, or variables in compiler options.
`constant width`	Words shown in constant-width font in the text are keywords and identifiers in C. All examples are written in constant-width font to indicate computer output.

Related Publications

The following C reference books supplement the contents of this manual:

- *The C Programming Language* by Brian Kernighan and Dennis Ritchie (Prentice-Hall, Inc., Englewood Cliffs, New Jersey: 1978)
- *A C Reference Manual* by Samuel P. Herbison and Guy L. Steele, Jr. (Prentice-Hall, Inc., Englewood Cliffs, New Jersey: 1984)

The following man pages, appearing in this binder, also supplement this manual:

- **cc**(1) C compiler
- **pdbx**(1) interactive parallel symbolic debugger
- **ld**(1) link editor

The following man page appears in the *DYNIX/ptx Reference Manual*:

- **a.out**(4) assembler and link editor output

These books provide tutorial introductions to C:

- *The C Primer*, by Les Hancock and Morris Krieger (McGraw-Hill Book Company, New York: 1982)
- *C Programming Guide*, by Jack Purdam (Que Corporation, Indianapolis, Indiana: 1983)
- *Learning to Program in C*, by Thomas Plum (Plum Hall Inc., Philadelphia, Pennsylvania: 1983)

Chapter 1
DYNIX/ptx C

1.1 Introduction 1-1

1.2 Supported Extensions 1-1
Structure Handling 1-2
Structure Assignment 1-2
Structure and Union Members 1-3
Functions Returning Structures 1-3
Structure Size 1-3
`enum` Type Specifier 1-3
`void` Type Specifier 1-4
Predefined Names 1-5
`signed` Type Specifier 1-6
`const` and `volatile` Type Specifiers 1-6
Using `const` and `volatile` with Pointers 1-7
`shared` and `private` Keywords 1-8

1.3 Porting Considerations 1-10
Register Variables 1-13

1.4 Running the Compiler 1-15

Figures

1-1 Valid and invalid uses of structure assignment 1-2
1-2 Example of the enum type specifier 1-4
1-3 Void type specifier 1-5
1-4 Example of a predefined name 1-5
1-5 Example of const referring to a single declaration 1-7
1-6 Example of const referring to an aggregate type 1-7
1-7 Test to determine type of assignment to unsigned short 1-11
1-8 Incorrect usage of asm statements in a C function 1-13

Tables

1-1 Using Modifiers with Pointers 1-8

Chapter 1
DYNIX/ptx C

1.1 Introduction

This manual describes the DYNIX/ptx C compiler. This compiler runs on Symmetry systems under the DYNIX/ptx operating system and follows the semantics and syntax of the C language as described in the first edition of Kernighan and Ritchie's *The C Programming Language*. The compiler also supports some extensions to K and R's definition, which are described in this chapter. The C compiler is not an ANSI compiler. For example, it does not support function prototyping or pointers to `void`. Any extension not discussed in this manual is not supported by the compiler.

1.2 Supported Extensions

This section describes the compiler's extensions. In addition to these extensions the compiler implements bitfields and register variables with some limitations. For more information on these features, refer to "Porting Considerations" in this chapter.

These extensions to the Kernighan and Ritchie book are implemented in accordance with the proposed ANSI C standard:

- structure handling
- `enum` type specifier
- `void` type specifier
- predefined names

The following extensions to previous versions of Sequent C are implemented in accordance with the proposed ANSI C standard:

- `signed` type specifier
- `const` type specifier
- `volatile` type specifier

The following Sequent C extensions support parallel programming and are nonstandard:

- `shared` keyword
- `private` keyword

1.2.1 Structure Handling

The compiler performs structure handling with the following improvements:

- implements structure assignment
- allows reuse of structure and union member names
- handles interruption of structure-returning functions
- does not limit structure size

Structure Assignment

Structures may be assigned, passed as arguments to functions, and returned by functions. The types of participating operands must be the same (must have the same structure tag). Other possible operations, such as equality comparison, have not been implemented. Figure 1-1 shows examples of both valid and invalid structure assignment.

```
struct a {int a[20];} sta1,sta2;
struct b {int a[20];} stb1,stb2;
struct a *stpt;
     sta1 = sta2;          /* legal   */
     stb1 = sta1;          /* illegal */
     *stpt = sta1;         /* legal   */
if (sta1 == sta2);         /* illegal */
```

Figure 1-1. Valid and invalid uses of structure assignment.

Structure and Union Members

Names of structure and union members need be unique only in the structure or union in which the member is located. Identical names for members of different structures or unions can be used in the same module.

Functions Returning Structures

For functions that return structures, the compiler requires modules that call these functions to declare the functions as returning structures. Although the return value may be ignored, failure to declare such a function correctly may cause the function to scramble the parameters.

A function that returns a structure cannot be declared as `void` even if the function's return value is ignored; the function must be declared as returning a structure wherever it is used or defined. Such a function may be coerced to `void` type after it is correctly declared.

Structure Size

The compiler places no limitation on the size of a structure. A structure of any size may be created, assigned, and referenced.

1.2.2 `enum` Type Specifier

The identifiers in an enumerator list are declared as constants that have type `int` and may appear wherever such constants are permitted. Enumeration types, which are analogous to the scalar types of Pascal, allow the user to name lists of identifiers.

Here is the syntax of the `enum` type specifier *enum-specifier*:

enum-specifier:
    enum *{ enum-list }*
    enum *identifier { enum-list }*
    enum *identifier*

enum-list:
    enumerator
    enum-list , enumerator

enumerator:
    identifier
    identifier = constant-expression

Identifier in the *enum-specifier* names a particular enumeration. Figure 1-2 shows an example that makes *color* the enumeration tag of a type describing various colors then declares *cp* to be a pointer to an object of that type. This example declares *col* to be an object of type *color*.

```
enum color { yellow, green, blue };
.
.
enum color *cp, col;
```

Figure 1-2. Example of the enum type specifier.

The *identifiers* in the *enumerator* are declared as constants and may appear wherever constants are required. If no *identifiers* with = appear, then the values of the constants begin at 0 and increase by 1 as the declaration is read from left to right. An *enumerator* with = gives the associated *identifier* the value indicated; subsequent *identifiers* continue the progression from the assigned value.

All enumeration tags and constants must be distinct and, unlike structure tags and members, are drawn from the set of identifiers.

Note that **lint** flags some enumeration type mismatches.

1.2.3 `void` Type Specifier

The `void` type specifier declares that a function has no return value or that the return value is discarded. It may be used in a cast or in a function declaration. Figure 1-3 shows an example of the `void` type specifier.

```
struct s function2()
{
    struct s r;
    r.a = r.b = 0;
    return r;
}
main()
{
    (void) function2();
}
```

Figure 1-3. Void type specifier.

1.2.4 Predefined Names

The following names are predefined by the C preprocessor to have the value 1:

- `sequent`
- `i386`
- `i387` or `w1167`
- `unix`

`i387` is undefined; instead, the predefined name `w1167` is defined to have the value 1. Figure 1-4 shows an example of using a predefined name.

```
#ifdef sequent
x = 42;    /* compiled */
#else
x = 0;    /* not compiled */
#endif
```

Figure 1-4. Example of a predefined name.

1.2.5 `signed` Type Specifier

The `signed` type specifier is a modifier to other type specifiers. If you specify `signed` alone, the object you are defining is considered to be of type `int`. The `signed` type specifier cannot modify a structure, union, enumeration specifier, or a `typedef` name. The following types can be modified by `signed`:

- `signed int`
- `signed short`
- `signed short int`
- `signed long`
- `signed long int`
- `signed char`
- `signed float`

The compiler accepts the ANSI `signed` type specifier even though in DYNIX/ptx C all the above types are `signed` by default. You can use the `signed` type specifier in code that you may port to other systems.

1.2.6 `const` and `volatile` Type Specifiers

The `const` and `volatile` type specifiers are modifiers of other type specifiers. If you specify either `const` or `volatile` alone, the object you are defining is considered to be of type `int`.

The `const` type specifier defines an object that cannot be modified. The compiler may put such values in read-only memory.

The `volatile` type specifier defines an object that you want to access in ways that may not be obvious to the compiler. Memory-mapped I/O registers, variables shared with other processes, and variables accessed in signal or interrupt handlers are examples of objects that should be declared `volatile`. When you declare an object `volatile`, the compiler keeps the variable in accessible memory, refraining from moving the object to a register even if the move would optimize a code segment.

For example, device status registers are typically polled repeatedly to wait for a physical I/O operation to complete. Without the `volatile` declaration, the optimizer might copy the device register into a CPU register once and use the CPU register subsequently without updating it with the current memory contents.

The specifiers `const` and `volatile` can refer either to a single declaration or to an aggregate type. Figure 1-5 shows an example of *const* referring to a single declaration. The sample code declares *alpha* as a `const` object, but not *beta*.

```
const struct s {int a,b;} alpha;
struct s  beta;
```

Figure 1-5. Example of const referring to a single declaration.

To make an aggregate type `const`, use a `typedef` declaration. Figure 1-6 shows an example that declares both *alpha* and *beta* as `const`.

```
typedef const struct s {int a,b;} consttype;
consttype alpha;
consttype beta;
```

Figure 1-6. Example of const referring to an aggregate type.

Using `const` and `volatile` with Pointers

Objects of type `const` and `volatile` have unique qualities compared to other type specifiers and are treated with special consideration by the compiler. It is important to make sure all pointers are declared properly when you use pointers to address `const` or `volatile` objects.

It is possible, using an explicit cast, to convert a pointer to a `const` object into a pointer to a type without the `const` attribute. Depending on the circumstances, the compiler may believe that the object is not a `const` and may allow the object to be modified. Make sure the type specifiers of pointers agree with the types of the objects they address.

You should not use a pointer to a `volatile` object unless it is declared as pointing to a `volatile` object. The compiler allows this, but by pointing to a `volatile` object with an improperly declared pointer you may defeat the purpose of defining the object `volatile`. The compiler makes assumptions for code optimization based on the declarations of the objects involved. For example, an object the compiler believes is nonvolatile may be put in a register to optimize the performance of a loop. This would prevent an external process from accessing the correct value of the `volatile` object.

Table 1-1 lists the possible methods of using `const` and `volatile` modifiers with pointers, and describes the results.

Table 1-1
Using Modifiers with Pointers

Declaration	Description
`const int *` `int * const`	(variable) pointer to a constant int constant pointer to a (variable) int

`volatile int *`
 	
`int * volatile`
	(nonvolatile) pointer to a volatile int; the integer can be changed by outside sources
	volatile pointer to a (nonvolatile) int; the pointer can be changed by outside sources

1.2.7 `shared` and `private` Keywords

Programs that use the `shared` keyword or the **-Y** option must be linked with the parallel library, */usr/lib/libpps.a*. (Refer to Sequent's *Guide to Parallel Programming* for information on the parallel programming library and for programming examples that use the `shared` and `private` keywords.)

The `shared` keyword declares that a variable is shared among all the parallel processes forked by the program. The `private` keyword declares that a variable cannot be shared and that each forked parallel process therefore has its own copy of the variable. By default, external and static variables that are not declared as shared are private. You can reverse this default with the **-Y** compiler option. In assembly language terms, shared variables are placed in the `.shdata` section of the object code and private variables are placed in the `.data` section. (For information on these

sections, refer to the *DYNIX/ptx Assembly Language User's Manual.*)

The `shared` and `private` keywords modify the storage class of a variable. Variables declared with the `shared` and `private` keywords must also fit into the `external` or `static` storage class. (Shared and private variables cannot appear in a `register` declaration.) The `shared` keyword can appear in the following constructs. (In these constructs, the parameter *type* stands for a type declaration.)

`shared` *type var*	Declares *var* to be a shared variable. This statement must appear outside a procedure. If it appears inside a procedure, the compiler generates an error message.
`shared` *type var=n*	Declares *var* as a shared variable and initializes its value to *n*. This statement must also appear outside any procedure or the compiler generates an error message.
`extern shared` *type var*	Declares that *var* is an external shared variable. This statement may appear inside or outside a procedure.
`static shared` *type var*	Declares that *var* is a static shared variable. This statement may appear inside or outside a procedure.
`static shared` *type var=n*	Declares that *var* is a static shared variable and initializes its value to *n*. This statement may appear inside or outside a procedure.

The `private` keyword can appear in the following constructs:

`extern private` *type var*	Declares that *var* is an external private variable.
`static private` *type var*	Declares that *var* is a static private variable.
`static private` *type var=n*	Declares that *var* is a static private variable and initializes its value to *n*.

NOTE

If a pointer appears in a `shared` *declaration, the pointer itself is shared, not the data it points to. For example, the statement* `shared char *p` *indicates that* `p` *itself is shared.*

If you are porting programs from other UNIX systems and your code uses the words `private`, `shared`, `signed`, `const`, or `volatile` as identifiers, the compiler produces syntax error messages. A simple solution that does not involve changing the source code is to use the preprocessor to redefine these variable names to names that do not conflict with Sequent keywords. This can be done on the command line with the **-D** switch.

1.3 Porting Considerations

When porting a C program to a Sequent system from another UNIX system, you may encounter programming errors that cause the program to fail on the Sequent system even though it apparently ran correctly on the other system. For example, in the VAX architecture, a value of type `float` is the same as the first four bytes of the corresponding `double` value. This is not true on the Sequent system, which conforms to the IEEE standard 754-1985, *Standard for Binary Floating-Point Arithmetic*. Consequently, a program that confuses `float` and `double` values may run correctly on a VAX but fail on the Sequent system. Use the **lint** utility to detect errors of this type.

In DYNIX/ptx C, the basic data types have the following sizes:

8 bytes for doubles
4 bytes for floats
4 bytes for ints and longs
4 bytes for pointers
2 bytes for shorts
1 byte for chars

The compiler allows both signed and unsigned variants of all integer data types. The compiler does not limit the size of structures in assignments. A program can assign any structure that can be declared.

However, the compiler can generate unexpected code for comparisons if an embedded assignment to a short stores an unsigned expression. This behavior is compatible with previous compilers, but it is not compatible with the proposed ANSI C standard. The proposed ANSI C standard requires value preserving in making an assignment to an `unsigned short`. The result of such an assignment would be an `int`. The DYNIX/ptx C compiler uses unsigned preserving. This assignment would result in an `unsigned int`. Figure 1-7 shows a test to determine which method is being used.

```
unsigned short data;
unsigned short * up = &data;
short *z = (short *) &data;
main ()
{
    /*
     * If the value at *z becomes -3, which is signed
     * and less than zero, the compiler is value
     * preserving.
     */
    if ( (*z += *up - 3) < 0 )
        printf (" Value preserving\n");
    else
        printf (" Unsigned preserving %d\n", *z);
}
```

Figure 1-7. Test to determine type of assignment to unsigned short.

To work around this, you should cast the expression as follows:

```
(*z += (short) *up - 3)
```

Pointers are 32-bit (4-byte) addresses.

The alignment of data (structure fields or entire variables) is as follows:

`char`	Aligned to 1 byte
`short`	Aligned to 2 bytes
`int`	Aligned to 4 bytes
`long`	Aligned to 4 bytes
`float`	Aligned to 4 bytes
`double`	Aligned to 4 bytes
`struct`	In an array of structures, if the base structure exceeds 2 bytes in size, each element of the array is aligned to 4 bytes. One-byte structures are aligned to 1 byte, and 2-byte structures are aligned to 2 bytes.
`bitfield`	If the bitfield is 17 or more bits and doesn't fit into the word currently being packed, it is aligned as if it were of type `int`; if the bitfield is 9 to 16 bits and doesn't fit into the word currently being packed, it is aligned as if it were of type `short`. The compiler does not sign extend or provide for true signed bitfields. Although bitfields can be declared `int` or `unsigned`, they are always positive.

The compiler treats `asm` statements as straight-line code. Any `asm` statements the compiler cannot reach are deleted. If C code appears between `asm` statements, branching between these statements is not allowed. Figure 1-8 shows an example of a function that will probably not produce the desired result.

```
foo ()
{
     int i = 0;

     asm ("label1:");
     foo (i++);
     asm ("  jmp    label1");
}
```

Figure 1-8. Incorrect usage of asm statements in a C function.

The `sizeof` operator yields an unsigned value. DYNIX 3.0 and Balance C compilers treat the result of this operation as a `signed int`. This fact could cause code ported from DYNIX-BSD or UNIX 4bsd to break.

The compiler has two different ways of controlling optimization of shared memory variables. By default the compiler assumes that you have declared as `volatile` all shared variables which are accessed or changed asynchronously by multiple processes. References to volatile variables (along with function calls) are considered to be synchronization points for shared variables. The compiler optimizes shared memory references as best it can, but it guarantees that the shared value is visible to all processes at synchronization points. Alternately, if any shared variables are to be accessed or changed asynchronously by multiple processes but are not declared as `volatile`, then you must ensure that the variables are correctly aligned in memory and compile your source with the **–i** option. This forces the optimizer to treat all shared variables as volatile, no matter what level of optimization (as specified with the **–O** compile-time option) is used.

1.3.1 Register Variables

The compiler implements three integer register variables and, for the **–fpa** option only, 12 double register variables. One of the three integer registers can hold character data. All three can hold data of type `short`, `integer`, or `pointer`.

If you explicitly assign register variables, the compiler honors the assignment. However, you may want to let the compiler optimize the register usage for you. Your code performance may be better if you do not explicitly assign register variables. The compiler uses any unassigned registers for code optimization, so even if you do assign some variables to registers, the compiler uses the remaining ones in optimizing your code.

You can deassign register variables without changing your source code. Invoke the compiler using the switch **-Dregister=** in your command line. The preprocessor substitutes a space for `register` each time it appears in your program. To implement this feature successfully, however, you must explicitly declare the type of the variable. For example, the legal declaration *register i;* will not compile with the **-Dregister=** switch. The keyword *register* disappears leaving only *i;*. You must declare this as *register int i;*.

There are some situations in which you will not want the compiler to put certain variables in registers even if your code is optimized by doing so. For example, you may want multiple processes to be able to access an object. You can direct the compiler not to optimize code by using registers in the following two ways:

- declaring certain variables as `volatile`
- using the **-i** flag on the command line

The `volatile` type specifier keeps the compiler from optimizing the declared objects. Using the `volatile` type specifier allows the generation of more compact and efficient code than does using the **-i** flag. Refer to the descriptions of the `volatile` type specifier and the **-i** flag for details on using these methods.

1.4 Running the Compiler

To invoke the compiler, use the following command line:

cc [*options*] *files* ...

files	Represents the names of C or assembly language source files to compile, and object files or libraries to link. A suffix of *.c* indicates a C source program. A suffix of *.s* indicates an assembly language source program. Any arguments without these file suffixes are assumed to be linker options, libraries of compatible routines, or object programs.
options	One or more valid options. Refer to the **cc**(1) man page in this binder for a list of options.

The compiler supports parallel programming extensions to the C language (refer to Sequent's *Guide to Parallel Programming*). It can create object, executable, or assembly language code according to the compiler options specified by the user.

C source programs are compiled, producing files of the same name but with the suffix *.o* substituted for *.c*. The *.o* file is normally deleted if a single C program is compiled and linked at the same time.

Assembly language source programs are assembled, producing *.o* files.

The C preprocessor options **–P**, **–E**, **–D**, **–U**, **–I**, and **–C** are recognized on the command line and passed to the C preprocessor. Refer to the **cpp**(1) man page in this binder for more detail about the C preprocessor.

For a description of the link-time options, refer to the **ld**(1) man page in this binder.

Appendix A
DYNIX/ptx C Language Calling Conventions

A.1 Calling Sequence A-1
A.2 Calling Conventions A-1
A.3 Argument List A-3

Appendix A
DYNIX/ptx C Language Calling Conventions

A.1 Calling Sequence

The psx/C calling sequence pushes arguments on top of the stack, then executes a call instruction. The calling functions push arguments on the stack in the following order: last argument first, first argument last.

A.2 Calling Conventions

Elements of the psx/C calling sequence are described as follows.

Argument Layout. Because the arguments are pushed onto the stack in last-to-first order, the first (leftmost) argument is always at a constant offset from the frame pointer register (`%ebp`), regardless of how many arguments were passed.

Argument Packing. Characters, ints, and short values use a double word (four bytes) and are aligned on a four-byte boundary. Floating-point values use eight bytes and are aligned on a four-byte boundary. Structures use a multiple of double words.

psx/C argument packing is compatible with the VAX UNIX 4.2bsd and Balance portable C compilers, so software written for either can be easily transported.

Order of Evaluation. The compiler evaluates and pushes arguments on the stack in last-to-first order.

Stack Alignment. Each function assumes that the stack is double-word aligned, so interrupt routines and assembly language routines must align the stack to a double word before exiting. Argument lists use a whole number of double words.

Saving Registers. The scratch registers for the i386 are `%eax`, `%ecx`, and `%edx`. For the Weitek 1167 (selected with the **–fpa** option) they are `%fp2` through `%fp7`. A function can change the contents of any scratch register without saving its previous value.

User registers are assigned to selected variables with the `register` keyword. For the i386 these registers are `%ebx`, `%edi`, and `%esi`. For the Weitek 1167 they are `%fp8` through `%fp31`. If a function uses a user register, the function must save the register's original value and restore it before returning to the caller.

The i387 (selected when **–fpa** is not specified) has an 8-register stack rather than scratch or user registers. (Refer to the *DYNIX/ptx Pdbx User's Manual* for a description of the i387 stack model.) Upon entry to a function, this stack must be empty. Upon exit the stack must be empty unless the function returns a floating point value. In this case the value is returned in stack register %st(0) and the remainder of the stack is empty.

NOTE

Calls to a floating point function where the caller and callee disagree on the function's type can cause an i387 stack underflow or overflow. An underflow occurs if the caller tries to pop %st(0) but the callee exited with the stack empty. An overflow occurs if the caller repeatedly pushes its return value onto the stack and the caller never pops it off.

Return Values. An integer or a pointer that is returned from a function is returned in register `%eax`. A floating-point value that is returned from a function is returned in i387 register `%st(0)` unless the **–fpa** compiler option is specified, in which case it is returned in register pair `%fp2`/`%fp3`.

If a function returns a structure, the calling function will pass an additional argument in register `%eax`. This argument points to where the called function is to return the structure. The called function copies the structure into the specified location during execution of the `return` statement. A function that returns a structure must also restore register `%eax` to the value that it contained upon entry to the funcion. Note that functions that return structures or floating point values must be correctly declared as such even if the return value is ignored.

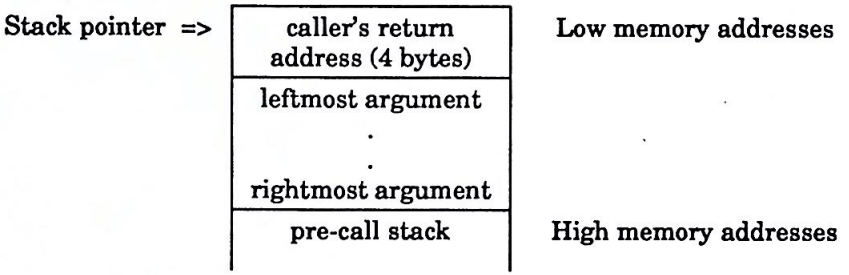
Passing Arrays. Arrays are passed by reference using their beginning addresses.

Passing Floating-Point and Integer Variables. The syntax of C does not allow the compiler to know the number or types of arguments of a called function. So that expressions can be used as arguments to functions, all floating-point numbers are passed as doubles and all integers are passed as long integers. This also simplifies writing functions that take a variable number of parameters, such as `printf`.

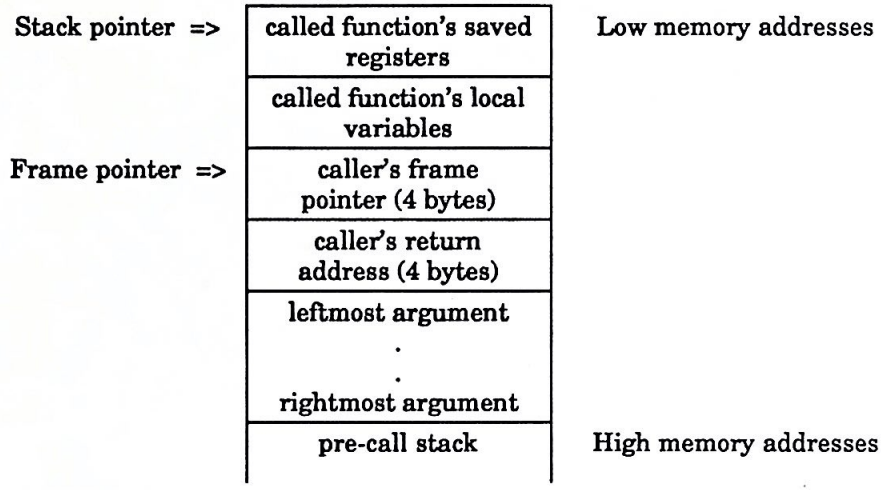
Popping Arguments Off the Stack. The caller, not the called function, is responsible for popping any arguments off the stack after the call has been completed.

A.3 Argument List

After a C routine has pushed the appropriate parameters and called another routine, the layout of the stack looks like this:



After the called routine's entry code is executed, the layout of the stack looks like this:



When the called routine terminates (with a **ret** instruction), it leaves the argument list intact; the caller discards the argument list with an explicit **add** instruction to the stack pointer.

Appendix B
Assembly Language Code in C Programs

B.1 Assembly Language Directives B-1

B.2 Assembly Language Macros B-2

Figures

B-1 Example of an asm statement B-1
B-2 Example of an asm macro definition B-3
B-3 Using the asm macro in a C program B-4
B-4 Using patterns to specify different storage modes B-5

Tables

B-1 Generated Assembly Code B-4

Appendix B
Assembly Language Code in C Programs

B.1 Assembly Language Directives

This compiler allows assembly language statements to be embedded in C programs through the use of the `asm` statement. An `asm` statement has the form:

`asm ("`*assembly code*`");`

where *assembly code* is one assembly language statement. Figure B-1 shows an example of an `asm` statement:

```
main()
{
      printf("hello");
      asm("nop");
}
```

Figure B-1. Example of an asm statement.

NOTE

`asm` *statements that appear between functions in the source code will not appear between the functions when the program is compiled.*

B.2 Assembly Language Macros

The compiler also allows assembly language macros to be expanded into assembly language in C programs. `asm` macros must be defined (or must be identified with an `#include` statement) in the source files that use them. All `asm` macros must be defined before any are invoked. An `asm` macro may be redefined for different source files.

An `asm` macro definition is of the form:

```
asm [type-specifier] identifier ([parameter-list])
{
[storage-mode-specification-line
        asm-body]...
}
```

where the *parameter-list* is a list of identifiers, excluding `asm` macros, and *storage-mode-specification-line* is defined as:

```
%[storage-mode [identifier [, identifier...]];...]
```

NOTE

The percent sign (%) that begins a pattern must appear in column one.

The following storage modes are recognized by the compiler in `asm` macros:

- *reg* – a compiler-selected temporary register or a C register variable that the compiler has allocated in a machine register
- *con* – a compile-time constant
- *mem* – any allowed machine addressing mode, including *reg* and *con*

Figure B-2 shows an example of an `asm` macro definition for the macro `ex1`.

```
asm void ex1(newpri)
{
%reg newpri;
    pushl  newpri
%con newpri;
    movl   newpri,%eax
    pushl  %eax
%mem newpri;
    addl   newpri,%eax
}
```

Figure B-2. Example of an asm macro definition.

The definition lines beginning with "%" are called *patterns*. If the arguments at the time the macro is called match the storage modes in a pattern, the code that follows the pattern line will be expanded.

If the arguments do not match any pattern, the expansion becomes a call to an external function with the same name. If you want the compiler to flag an error rather than call a function if the arguments don't match any pattern, include the following line at the end of your `asm` function declaration:

```
%error
```

Figure B-3 shows how the `ex1` macro may be used. The code shown in Figure B-2 is placed before the code shown in Figure B-3 in the source file.

```
main(){
    register int x;
    int y;
    ex1(x);
label1:
    ex1(y);
label2:
    ex1(7);
}
```

Figure B-3. Using the asm macro in a C program.

Table B-1 demonstrates the assembly code generated by the compiler for the uses of `ex1` in the above example.

Table B-1
Generated Assembly Code

code..	matches..	generates..
ex1(x);	%reg	pushl %edi
ex1(y);	%mem	addl -4(%ebp), %eax
ex1(7);	%con	movl $7, %eax pushl %eax

The first use of `ex1` has a register variable as its argument. This argument has a storage mode that matches *reg*, the storage mode in the first pattern. Therefore, the compiler expands the first code body. Note that *newpri*, the formal parameter in the definition, has been replaced in the expanded code by the compiler's idea of the assembly-time name for the variable *x*, namely `%edi`. The other calls to `ex1` are expanded similarly.

A pattern may specify different storage modes for different arguments. Figure B-4 shows an example of this.

```
asm void ex2(a,b)
{
%reg a; con b;
    movl   b,a
%mem a; reg b;
    addl   a,b
}
```

Figure B-4. Using patterns to specify different storage modes.

CAUTION

`asm` *macros should access only scratch registers, global data, and actual parameters to the* `asm` *macro. Embedded assembly code that "knows" register conventions or stack layout and uses this "knowledge" to access data is dangerous.*

Index

/usr/lib/libpps.a 1-10
–Dregister= compiler option 1-15
–i compiler option 1-15
–Y compiler option 1-10

A

Alignment
  of arguments A-1
  of bitfields 1-13
  of data 1-13
  of stack A-1
ANSI standard compliance 1-1
Arguments
  layout A-1
  list A-3
  on stack A-3
  order of evaluation A-1
  packing A-1
  returned from a function A-2
Arrays
  passing A-3
asm directive B-1
asm statements 1-14
Assembly language
  in C programs ..... Appendix B
  macros ..... B-2 to 5
Assignment
  of structures 1-2

B-C

Bitfields
  packing 1-13
C calling conventions ..... Appendix A
C preprocessor 1-15,16

C standard (K&R)
enhancements to 1-1
Calling conventions ..... A-1 to 3
Calling sequence ..... Appendix A
elements of A-1
cc command 1-16
char data type 1-12
Compiler
invocation 1-16
options
-Dregister= 1-15
-i 1-15
-Y 1-10
compiler keywords
and parallel programming 1-10
const type specifier ..... 1-7 to 8
with pointers ..... 1-8 to 10
cpp 1-15,16

D

Data alignment 1-13
Data types
char 1-12
double 1-12
float 1-12
int 1-12
long 1-12
short 1-12
storage for 1-12
Directive
assembly language B-1
double data type 1-12

E

Enhancements
to C standard (K&R) 1-1
enum type specifier ..... 1-3 to 4
Enumeration type mismatches
and lint 1-4
Enumeration type specifier ..... 1-3 to 4

Evaluation of arguments
order of A-1
external storage class 1-10

F-G-H

float data type 1-12
Floating-point variables
passing A-3
Functions
returning structures 1-3

I

i386 1-6
i387 1-6
int data type 1-12
Integer variables
passing A-3
Invocation
of compiler 1-16

J-K

Keywords
private ..... 1-10 to 11
shared ..... 1-10 to 11
with external or static storage class 1-10

L-M-N-O

lint
enumeration type mismatches and 1-4
long data type 1-12
Macros
assembly language, in C programs ..... B-2 to 5
Names
predefined 1-6
Optimization
and register variables 1-15
Order of argument evaluation A-1

P

Packing, of arguments A-1
Parallel programming
  and compiler keywords 1-10
Passing arrays A-3
Passing floating-point variables A-3
Passing integer variables A-3
pattern ..... B-2 to 3
Popping arguments off the stack A-3
Porting considerations ..... 1-11 to 14
Predefined names 1-6
  `i386` 1-6
  `i387` 1-6
  `sequent` 1-6
  `unix` 1-6
  `w1167` 1-6
`private` keyword ..... 1-10 to 11
  and storage class 1-10

Q-R

Register variables 1-15
Registers
  saving on the stack A-2
Return value
  and `void` type specifier 1-4
  discarded 1-4
Return values A-2
Running the compiler 1-16

S

Saving registers A-2
`sequent` 1-6
`shared` keyword ..... 1-10 to 11
  and storage class 1-10
`short` data type 1-12
Sign extension
  and bitfields 1-14
`signed` type specifier ..... 1-6 to 7
`sizeof operator` 1-14

Stack
popping arguments off A-3
Stack alignment A-1
`static` storage class 1-10
Storage for data types 1-12
storage modes B-2
Structures ..... 1-2 to 3
assignment of 1-2
limitation in size of 1-3
returned by functions 1-3
union members and 1-3

T

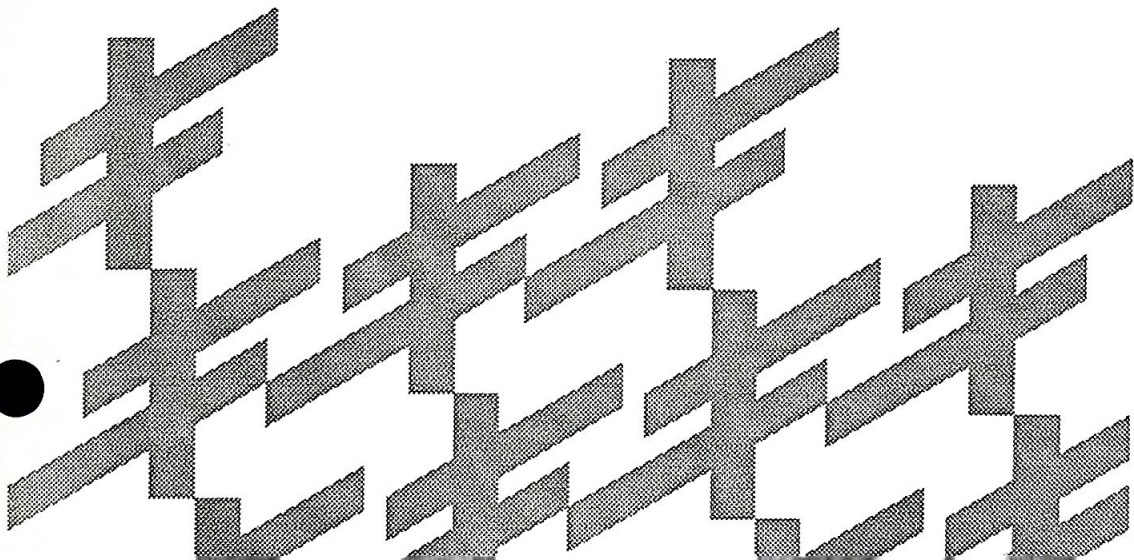
Type specifiers
`const` ..... 1-7 to 8
with pointers ..... 1-8 to 10
`enum` ..... 1-3 to 4
`signed` ..... 1-6 to 7
`void` ..... 1-4 to 5
`volatile` ..... 1-7 to 8,15
with pointers ..... 1-8 to 10

U-V-W-X-Y-Z

Union members
structures and 1-3
`unix` 1-6
Variables
register 1-15
`void` type specifier ..... 1-4 to 5
`volatile` type specifier ..... 1-7 to 8,15
with pointers ..... 1-8 to 10
`wl167` 1-6

DYNIX/ptx®
lint Reference Manual

1003-49927-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

IBM is a registered trademark of International Business Machines Corp.

Copyright 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America

DYNIX/ptx lint Reference Manual

1 Introduction 1

2 Running lint 1
lint Libraries 3
Using Default lint Libraries 3
Creating a lint Library 4

3 lint Message Types 4
Unused Variables and Functions 5
Set/Used Information 6
Flow of Control 6
Function Values 7
Type Checking 8
Type Casts 9
Nonportable Character Use 9
Assignments of **longs** to **ints** 10
Unusual Constructions 10
Old Syntax 11
Pointer Alignment 12
Multiple Uses and Side Effects 12
Parallel Programming Support 13

DYNIX/ptx lint Reference Manual

1 Introduction

The **lint** program examines C language source programs and library specifications, checking them for consistency. **lint** performs the following tasks:

- checks for bugs
- enforces portability restrictions
- checks for elements that are legal but wasteful or error-prone
- enforces the C language type rules

2 Running lint

The **lint** command has the following form:

lint [*–options*] *file* [...] [*lint-library* ...]

options	One or more valid options from the following list.
file	C source code files to be checked by **lint**. C source files must end with *.c*.
lint-library	**lint** library files to be used in checking the C source code. **lint** library files end with *.ln*.

The command line options are as follows:

–a	Suppress messages about assignments of **long** values to variables that are not `long`.

–b Suppress messages about break statements that cannot be reached.

–c Check only for intra-file bugs; leave external information in files suffixed with *.ln*.

–h Do not apply heuristics which attempt to detect bugs, improve style, and reduce waste.

–l *name* Include additional lint library *llib-lname.ln*. For example, you can include a **lint** version of the math library by inserting **–lm** on the command line. This argument does not suppress the default use of *llib-lc.ln*. These **lint** libraries must be in the assumed directory. This option can be used to reference local **lint** libraries and is useful in the development of multi-file projects.

–n Do not check for compatibility with either the standard or the portable **lint** library.

–o *name* Create a lint library from input files named *llib–lname.ln*. The **–c** option nullifies any use of the **–o** option. To produce a *llib-lname.ln* library without extraneous messages, we recommend that you use the **–x** option. The **–v** option is useful if the source files are just external interfaces.

–p Attempt to check portability to the IBM and GCOS dialects of C. In addition to stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–u Suppress messages about function and external variables used and not defined, or defined and not used.

–v Suppress messages about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

–Wc,–seq Recognize the Sequent keywords: *const*, *volatile*, *signed*, *shared*, and *private*.

–Wc,–fpa **lint** code intended for the floating-point accelerator. This option has an effect only if the **cpp**(1) macros **i387** or **w1167** were used for conditional compilation.

When specifying more than one option, combine them into a single argument, such as **-ab** or **-xha**.

lint also recognizes the **-D**, **-U**, and **-I** options of **cpp**(1) and the **-g**, **-O**, and **-Wc** options of **cc**(1) as separate arguments. The **-g**, **-O**, and **-Wc** options are ignored, but, by recognizing these options, **lint**'s behavior is closer to that of the **cc**(1) command. Other options are warned about and ignored.

The pre-processor symbol *lint* is defined to allow certain questionable code to be altered or removed for **lint**. Therefore, the symbol *lint* should be thought of as a reserved word for all code that is planned to be checked by **lint**.

2.1 lint Libraries

lint library files have the file extension *.ln* and begin with this comment:

```
/* LINTLIBRARY */
```

This comment is followed by a series of dummy function definitions. The critical parts of these definitions are the declaration of the function return type, whether the dummy function returns a value, and the number and types of arguments to the function. This information is used by **lint** when it is checking source code.

lint libraries are created from C library source code. They are processed almost exactly like ordinary source files. The only difference is that functions which are defined in a library file, but are not used in a source file, do not cause **lint** to issue messages. **lint** does not simulate a full library search algorithm, and prints messages if the source files contain a redefinition of a library routine. You can use the default **lint** libraries installed on your system or you can create your own **lint** libraries.

Using Default lint Libraries

You can use standard **lint** libraries installed on your system by specifying the library on the **lint** command line in the style of the link editor (**ld**) **-l** option. For example, **lint** accepts arguments like the following:

-lm
-lseq
-lpps

lint checks the source code for compatibility with these libraries by accessing library description files whose names are constructed from the library arguments.

If you want your code checked against the library specifications you compiled with, be sure to specify these libraries when using **lint**. For example, if you compiled your C program with this command:

```
cc a.c b.c d.c -lseq
```

you would **lint** your program with the following command:

```
lint a.c b.c d.c -lseq
```

Using **-lseq** specifies the Sequent library; likewise, using **-lm** or **-lpps** specifies the math library or the Parallel Programming Library.

Creating a lint Library

You can create your own **lint** libraries with **lint** by specifying the **-o** option followed by the C source routines or libraries you want checked. In the following example, three C routines are input and the output **lint** library name is specified as *my-lint-lib*:

```
lint -o my-lint-lib routinea.c routineb.c routined.c
```

The output **lint** library filename is *llib.lmy-lint-lib.ln*. **lint** automatically adds the prefix *llib.l* and the suffix *.ln* to the output name you specified with the **-o** option.

To use your **lint** library, specify it on the command line as follows:

```
lint a.c b.c d.c llib.lmy-lint-lib.ln
```

If you don't specify any libraries, either with **-l** or by using a *.ln* file on the command line, **lint**, by default, checks the programs it is given against a standard library file that contains descriptions of the programs that are normally loaded when a C language program is run. When the **-p** option is used, another file is checked containing descriptions of the standard library routines which are expected to be portable across various machines. The **-n** option can be used to suppress all library checking.

3 lint Message Types

The following paragraphs describe the major categories of messages printed by **lint**.

3.1 Unused Variables and Functions

lint prints messages about variables and functions which are defined but not otherwise used, unless the message is suppressed with the **–u** or **–x** option.

As programs evolve, previously used variables and arguments to functions may become unused. It is not uncommon for external variables or even entire functions to become unnecessary and yet not be removed from the source. These errors cause inefficiency and make programs harder to understand and change. Also, information about such unused variables and functions occasionally uncovers bugs.

Some programming styles permit a function to be written with an interface where some of the function's arguments are optional. Such a function can be designed to accomplish a variety of tasks depending on which arguments are used. Normally **lint** prints messages about unused arguments; however, specifying the **–v** option suppresses the printing of these messages. When **–v** is in effect, no messages are produced about unused arguments except for those arguments which are unused and also declared as register arguments. This prevents waste of register resources.

Messages about unused arguments can be suppressed for one function by adding the following comment to the source code before the function. It has the effect of the **–v** option for only one function.

```
/* ARGSUSED */
```

The following comment can be used to suppress messages about variable numbers of arguments in calls to a function. The comment should be added before the function definition.

```
/* VARARGS */
```

In some cases, it is desirable to check the first several arguments and leave the later arguments unchecked. This can be done by adding a digit giving the number of arguments which should be checked. For example, the following causes only the first two arguments to be checked:

```
/* VARARGS2 */
```

If **lint** is applied to some but not all files of a group that are to be loaded together, **lint** complains about unused or undefined variables. This information is more distracting than helpful since functions and variables that are defined may not be used; and, conversely, functions and variables defined elsewhere may be used. The **–u** option suppresses the spurious messages.

3.2 Set/Used Information

lint tries to detect cases where a variable is used before it is set. **lint** detects local variables (automatic and register storage classes) whose first use appears physically earlier in the input file than the first assignment to the variable. **lint** assumes that taking the address of a variable constitutes using a variable, since the actual use may occur at any later time, in a data-dependent fashion.

The restriction to the physical appearance of variables in the file makes the algorithm very simple and quick to implement since the true flow of control need not be discovered. It does mean that **lint** can print error messages about program fragments that are legal, but that would be considered stylistically bad. Because static and external variables are initialized to zero, no meaningful information can be discovered about their uses. However, **lint** does deal with initialized automatic variables.

The set/used information also permits recognition of local variables that are set and never used. These are often inefficient and may also cause bugs.

3.3 Flow of Control

lint attempts to detect unreachable portions of a program. It prints messages about unlabeled statements immediately following **goto**, **break**, **continue**, or **return** statements. **lint** attempts to detect loops that cannot be entered or exited, and to recognize the special cases **while**(1) and **for**(;;) as infinite loops. Valid programs may have such loops, but they are considered to be bad style. If you do not want messages about unreached portions of the program, specify **-b**.

lint does not detect functions that are called and never return. Thus, a call to **exit** may cause unreachable code that **lint** does not detect. The most serious effects of this are in the determination of returned function values (refer to "Function Values" later in this chapter). If a particular place in the program is thought to be unreachable in a way that is not apparent to **lint**, the following comment can be added to the source code at the appropriate place.

```
/* NOTREACHED */
```

This comment informs **lint** that a portion of the program cannot be reached, and **lint** does not print a message about the unreachable portion.

Programs generated by **yacc** and **lex** may have unreachable **break** statements, but messages about them are of little importance. There is usually nothing the user can do about them, and the resulting messages would clutter up the **lint** output. Invoking **lint** with the **-b** option suppresses the messages.

3.4 Function Values

Sometimes functions return values that are never used, or programs incorrectly use function values that have never been returned. **lint** addresses this problem in several ways.

Locally, within a function definition, the appearance of both of the following statements is cause for alarm:

```
return (expr);
```

and

```
return ;
```

lint issues this message:

```
function name has return(e) and return
```

The most serious difficulty with this is detecting when a function return is implied by the flow of control reaching the end of the function. This can be seen with the following example:

```
f(a){
        if(a) return(3);
        g();
        }
```

Notice that, if *a* tests false, **f** calls **g** and then returns with no defined return value; this triggers a message from **lint**. If **g**, like **exit**, never returns, the message is still produced when in fact nothing is wrong. The following comment in the source code causes the message to be suppressed:

```
/*NOTREACHED*/
```

In practice, serious bugs have been discovered by this feature.

On a global scale, **lint** detects cases where a function returns a value that is sometimes or never used. When the value is never used, it may be due to an inefficiency in the function definition that can be overcome by casting the return value of the function to type (void).

```
(void) fprintf(stderr,"File busy. Try again later!\n");
```

When the value is sometimes unused, it may represent bad style. An example of this is not testing for error conditions.

The opposite problem, using a function value when the function does not return one, is also detected. This is a serious problem.

3.5 Type Checking

lint enforces the C type checking rules more strictly than the C compiler. The additional checking is in four major areas:

- Across certain binary operators and implied assignments
- At the structure selection operators
- Between the definition and uses of functions
- In the use of enumerations

The assignment, conditional (**?:**), and relational operators have an implied balancing between types of the operands. The argument of a **return** statement and expressions used in initialization also have this property. In these operations, **char**, **short**, **int**, **long**, **unsigned**, **float**, and **double** types may be freely intermixed. The types of pointers must agree exactly except that arrays of *x*s can, of course, be intermixed with pointers to *x*s.

The type checking rules also require that, in structure references, the left operand of the **->** be a pointer to structure, the left operand of the **.** be a structure, and the right operand of these operators be a member of the structure implied by the left operand. Similar checking is done for references to unions.

Strict rules apply to function argument and return value matching. The types **float** and **double** may be freely matched, as may the types **char**, **short**, **int**, and **unsigned**. Also, pointers can be matched with the associated arrays. Aside from this, all actual arguments must agree in type with their declared counterparts.

lint checks to make sure that enumeration variables or members are not mixed with other types or other enumerations, and that the only operations applied are =, initialization, ==, !=, and function arguments and return values.

To turn off strict type checking for an expression, add the following comment to the source code immediately before the expression:

```
/* NOSTRICT */
```

This prevents strict type checking for only the next line in the program.

3.6 Type Casts

The type cast feature in C was introduced as an aid to producing more portable programs. Consider the following assignment where **p** is a character pointer:

```
p = 1 ;
```

lint prints a message as a result of detecting this.

Consider the assignment in which a cast has been used to convert the integer to a character pointer.

```
p = (char *)1 ;
```

Even though the programmer intended this, **lint** continues to print messages.

3.7 Nonportable Character Use

On some systems, characters are signed quantities with a range from –128 to 127. In other C implementations, characters can have only positive values. Thus, **lint** messages are issued that designate certain comparisons and assignments as being illegal or nonportable. For example, the following code fragment works on one machine, but fails on machines where characters always have positive values:

```
char c;
        ...
if( (c = getchar()) < 0 ) ...
```

The solution is to declare *c* as an integer since **getchar** is actually returning integer values. In any case, **lint** prints this message:

```
nonportable character comparison
```

Bit fields pose a similar problem. When assignments of constant values are made to bit fields, the field may be too small to hold the value. This is true because on some machines bit fields are considered as signed quantities. While it may seem logical to consider that a two-bit field declared of type **int** cannot hold the value 3, the problem is solved if the bit field is declared to have type **unsigned**.

3.8 Assignments of longs to ints

Bugs may arise from the assignment of a **long** to an **int**, which truncates the contents. This can happen in programs which have been incompletely converted to use **typedefs**. When a **typedef** variable is changed from **int** to **long**, the program may stop working because some intermediate results are assigned to **ints**, which are truncated. Specifying the **-a** option suppresses messages about the assignment of **longs** to **ints**.

3.9 Unusual Constructions

Some legal but unusual constructions are detected by **lint**. The messages encourage better code quality, clearer style, and may even point out bugs. Specifying the **-h** option suppresses these checks.

For example, in the following statement, the * does nothing.

```
*p++ ;
```

lint issues the following message:

```
null effect
```

The following program fragment results in a test that never succeeds:

```
unsigned x ;
if( x < 0 ) . . .
```

Similarly, the following test

```
if( x > 0 ) . . .
```

is equivalent to

```
if( x != 0 )
```

which may not be the intended action. In this case, **lint** issues the following message:

```
degenerate unsigned comparison
```

If a program contains something similar to the following:

```
if( 1 != 0 ) . . .
```

lint prints the following message because the comparison of 1 with 0 gives a constant result:

```
constant in conditional context
```

Another construction detected by **lint** involves operator precedence. Bugs which arise from misunderstandings about the precedence of operators can be accentuated by spacing and formatting, making such bugs hard to find. For example, the following statements probably do not do what was intended:

```
if( x&077 == 0 ) . . .
```

and

```
x<<2 + 40
```

The best solution is to parenthesize such expressions, and **lint** encourages this by an appropriate message.

3.10 Old Syntax

Several forms of older syntax are now illegal. Illegal syntax falls into two classes: assignment operators and initialization.

The older forms of assignment operators (for example, =+, =-) could cause ambiguous expressions, such as the following:

```
a =-1 ;
```

This expression could be taken as either

```
a =- 1 ;
```

or

```
a = -1 ;
```

The situation is especially perplexing if this kind of ambiguity arises as the result of a macro substitution. The newer and preferred operators (for example, +=, -=) have no ambiguities. To encourage the abandonment of the older forms, **lint** prints messages about the old operators.

A similar issue arises with initialization. The older language allowed the following syntax to initialize *x* to 1:

```
int x 1;
```

This also caused syntactic difficulties. For example, the initialization

```
int x ( -1 ) ;
```

looks somewhat like the beginning of a function definition:

```
int x ( y ) { . . .
```

The compiler must read past *x* in order to determine the correct meaning. Again, the problem is even worse when the initializer involves a macro. The current syntax avoids ambiguity by placing an equals sign between the variable and the initializer.

```
int x = -1 ;
```

3.11 Pointer Alignment

Certain pointer assignments may be allowed on some machines and illegal on others because of alignment restrictions. **lint** tries to detect cases where pointers are assigned to other pointers and alignment problems might arise. The following message results from this situation:

```
possible pointer alignment problem
```

3.12 Multiple Uses and Side Effects

In complicated expressions, the best order in which to evaluate subexpressions may be highly machine dependent. For example, on machines in which the stack runs backwards, function arguments are probably best evaluated from right to left. On machines with a stack running forward, left to right seems better. Function calls embedded as arguments of other functions may or may not be treated similarly to ordinary arguments. Similar issues arise with other operators that have side effects, such as with the assignment operators and the increment and decrement operators.

In order that the efficiency of C on a particular machine not be compromised, the C language leaves the order of evaluation of complicated expressions up to the local compiler. In fact, various C compilers have considerable differences in the order in which they evaluate complicated expressions. In particular, if any variable is changed by a side effect and also used elsewhere in the same expression, the result is explicitly undefined.

lint checks for the important special case where a simple scalar variable is affected. For example, the following statement causes **lint** to print a message calling attention to this condition:

```
a[i] = b[i++];
```

The message issued is this:

```
warning: i evaluation order undefined
```

3.13 Parallel Programming Support

If the environment has a variable *PARALLEL* with a value $N > 1$, **lint** will process N input files in parallel. Using this feature can substantially reduce the time to **lint** large programs having many input files.

Index

A

a option 1
Alignment
  pointer 12
ARGSUSED comment 5
Argument
  unused 2
Assignment
  `long` 1,10

B

b option 2
Break statements 2
Bugs
  detection of 2
  intra-file 2

C

c option 2
Cast feature 9
Character use 9
comment
  ARGSUSED 5
  NOSTRICT 9
  NOTREACHED 6,7
  VARARGS 5
Conditional
  compilation 2

D-E-F

Evaluating expressions 12
Floating-point accelerator 2
Flow of control 6,7

Function
unused 5

G-H-I-J-K

h option 2
i387 option 2
`int` data type 10
Keywords
Sequent 2

L

l option 2
lint library 2
lint library 1,2,3,4
creation of 4
default 3,4
lint, running 1
`long` data type 1,10

M-N

Message types 4
n option 2
Nonportability
character use 9
NOSTRICT comment 9
NOTREACHED comment 6,7

O

o option 2
Old
syntax 11,12
Options
a 1
b 2
c 2
cc 3
cpp 3
h 2

l 2
n 2
o 2
p 2
u 2
v 2
Wc,fpa 2
Wc,seq 2
x 2

P

p option 2
Parallel programs 13
Pointer
  alignment 12
Portability
  dialect 2

Q-R-S-T

Running **lint** 1
Sequent
  keywords 2
Specifying options 3
Syntax
  old 11,12
Type casts 9
Type checking 8

U

u option 2
Unreachable code 6,7
Unused
  functions 5
  variables 5
Unusual constructions 10,11

V

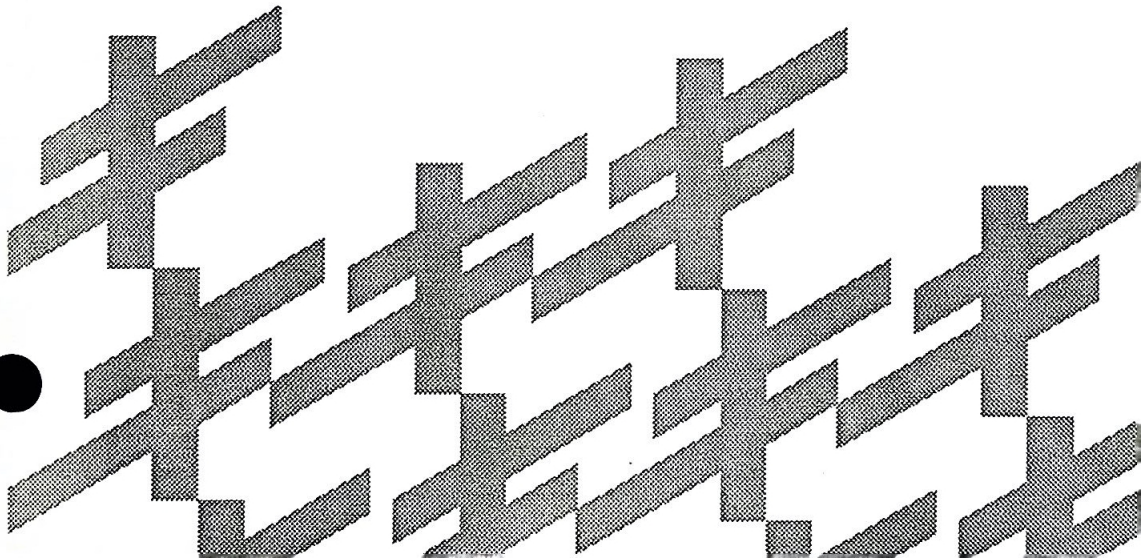
v option 2,5
VARARGS comment 5
Variable
  external 2
  function 2
  unused 2,5

W-X-Y-Z

w1167option 2
Wc,fpa option 2
Wc,seq option 2
x option 2

DYNIX/ptx®
ld User's Guide

1003-49928-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

Copyright 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America

DYNIX/ptx ld User's Guide

1 The Link Editor, ld 1
Running ld 2
Memory Configuration 2
Sections 3
Addresses 3
Binding 3
Object File 3

2 Link Editor Command Language 4
Expressions 4
Assignment Statements 6
Specifying a Memory Configuration 8
Section Definition Directives 9
File Specifications 10
Load a Section at a Specified Address 12
Aligning an Output Section 13
Grouping Sections Together 13
Creating Holes Within Output Sections 16
Creating and Defining Symbols at Link-Edit Time 17
Allocating a Section into Named Memory 19
Initialized Section Holes or .bss Sections 20

3 Notes and Special Considerations 22
Use of Archive Libraries 23
Nonrelocatable Input Files 25
Changing the Entry Point 25
Dealing With Holes in Physical Memory 26
Allocation Algorithm 27
Incremental Link Editing 28
DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections 30
Output File Blocking 31

4 Syntax Diagram for Input Directives 32

Figures

1 C code showing where code is stored in object file 4
2 Syntax of the SECTIONS directive 10
3 Specifying output section files 11
4 Loading a section in a specified address 12
5 Placing output sections in the same segment 15
6 Output sections placed in separate segments 15
7 Creating holes within output sections 16
8 Assigning an allocation-dependent value to a symbol 18
9 Assigning an allocation-independent value to a symbol 19
10 Allocating a section into named memory 20
11 Initializing section holes 22
12 Dealing with holes in physical memory 27
13 Creating and linking subsystems 29

Tables

1 Syntax Diagram for Input Directives 32

DYNIX/ptx ld User's Guide

1 The Link Editor, ld

This manual describes the link editor (**ld**) and the link editor command language. For the default operation of **ld**, refer to the **ld**(1) man page. The link editor command language enables you to override th defaults and perform these tasks:

- Specify the memory configuration of the target machine
- Combine the sections of an object file in arrangements other than the default
- Bind sections of an object file to specific addresses or within specific portions of memory
- Define or redefine global symbols

Usually there is no need to have control over object files and their location in memory. However, when you need control of the link editor output, you do it by means of the link editor command language described later in this manual.

NOTE

The capabilities of the link editor command language are needed in only a few rare circumstances. You would use the **ld** *command language when linking the kernel, stand-alone programs, or diagnostic programs. The* **ld** *defaults should be used in all other cases.*

1.1 Running ld

To run the link editor, use the following syntax:

ld [*–options*] *filename ...*

options	One or more valid options. Refer to the **ld**(1) man page in this binder for a list of options.
filename	Object files, archive library files, or text files (ifiles) containing link editor directives.

Link editor command language directives are passed in a file specified on the **ld** command line. This file is traditionally referred to as the *ifile*, but may have any name that will not be confused with an object file or archive library file. Therefore, any file named on the command line that is not identifiable as an object module or an archive library is assumed to contain directives.

1.2 Memory Configuration

Virtual memory is partitioned into configured and unconfigured memory. Configured memory can be allocated; unconfigured memory cannot. The default condition is to treat all memory as configured. However, some microprocessor applications may have different types of memory at different addresses. For example, an application might have 3K of PROM (Programmable Read-Only Memory) beginning at address 0, and 8K of ROM (Read-Only Memory) starting at 20K. Addresses in the range 3K to 20K–1 are not configured. Unconfigured memory is treated as reserved and is unusable by **ld**. Since nothing can be linked into unconfigured memory, specifying a certain memory range to be unconfigured is one way of marking the addresses in that range as illegal or nonexistent with respect to the linking process. Memory configurations other than the default must be explicitly specified by the user.

Throughout this manual, discussions of memory, addresses, and so on, refer to configured sections of the address space unless otherwise specified.

1.3 Sections

A section of an object file is the smallest unit of relocation and must be located in a contiguous block of memory. A section is identified by a starting address and a size. Information describing all the sections in a file is stored in section headers at the start of the file. Sections from input files are combined to form output sections that contain executable text, data, or a mixture of both. Although there may be holes or gaps between output sections, storage is allocated contiguously within each output section and may not overlap a hole in memory.

1.4 Addresses

The physical address of a section or symbol is the relative offset from address zero of the address space. The physical address of an object is not necessarily the location at which it is placed when the process is executed. For example, on a system with paging, the address is with respect to address zero of the virtual space, and the system performs another address translation.

1.5 Binding

It is often necessary to have a section begin at a specific, predefined address in the address space. The process of specifying this starting address is called binding, and the section in question is said to be bound to or bound at the required address. While binding is most commonly relevant to output sections, it is also possible to bind special absolute global symbols with an assignment statement in the **ld** command language.

1.6 Object File

Object files are produced both by the assembler, **as**(1) (usually as a result of calling **cc**(1)) and by **ld**. Relocatable object files are files whose addresses are not absolute and can therefore be relocated by the link editor. **ld** accepts relocatable object files as input and produces an output object file that may or may not be relocatable. Under certain special circumstances, the input object files given to **ld** can also be absolute files.

Compiler object files may contain, among others, sections called `.text` and `.data`. The `.text` section contains the instruction text (executable instructions), and `.data` contains initialized data variables. Figure 1 shows where information in a C program is stored in the object file. The sample code contains a global declaration (that is, not inside a function) and an assignment.

```
int i = 100;   /* stored in .data section
func() {
   i = 0;      /* stored in .text section
}
```

Figure 1. C code showing where code is stored in object file.

2 Link Editor Command Language

The following paragraphs describe the link editor command language that can be passed to **ld** in files (ifiles) specified on the **ld** command line.

2.1 Expressions

Expressions may contain global symbols, constants, and some of the basic C language operators. (See Table 6-1, "Syntax Diagram for Input Directives.") Constants are specified the same as in C. Numbers are recognized as decimal unless preceded with 0 for octal or 0x for hexadecimal. All numbers are treated as long integers. Symbol names may contain uppercase or lowercase letters, digits, and underscores (_). Symbols within an expression have the value of the address of the symbol only.

ld does not do symbol table lookup to find the contents of a symbol, the dimensionality of an array, or structure elements declared in a C program, and so forth. **ld** uses a **lex**-generated input scanner to identify symbols, numbers, operators, and so on. The following names are reserved and unavailable as symbol names or section names:

ADDR	**BLOCK**	**GROUP**	**NEXT**	**RANGE**	**SPARE**
ALIGN	**COMMON**	**INFO**	**NOLOAD**	**REGIONS**	**PHY**
ASSIGN	**COPY**	**LENGTH**	**ORIGIN**	**SECTIONS**	**TV**
BIND	**DSECT**	**MEMORY**	**OVERLAY**	**SIZEOF**	
addr	**bind**	**l**	**next**	**origin**	**s**
align	**block**	**len**	**o**	**phy**	**sizeof**
assign	**group**	**length**	**org**	**range**	**spare**

The following operators are supported and have the same meaning as in the C language. They are shown in order of precedence from high to low. Operators on the same line have the same precedence.

```
!  - (UNARY Minus)
*  /  %
+  - (BINARY Minus)
>>  <<
==  !=  >  <  <=  >=
&
|
&&
||
=  +=  -=  *=  /=
```

2.2 Assignment Statements

External symbols may be defined and assigned addresses with an assignment statement. The syntax of the assignment statement is

```
symbol = expression;
```

or

```
symbol op = expression;
```

where *op* is one of the operators +, –, *, or / . Assignment statements must be terminated by a semicolon.

All assignment statements (except the one case described in the next paragraph) are evaluated after allocation has been performed. This occurs after all symbols that are defined in input files are appropriately relocated. Therefore, when a symbol is used in an expression in an assignment statement, that symbol's address in the output file is used. References within text and data (to symbols given a value through an assignment statement) access this latest assigned value. Assignment statements are processed in the same order in which they are input to ld.

Assignment statements are normally placed outside the scope of section-definition directives (refer to "Section Definition Directives" later in this chapter). However, there exists a special symbol, called dot (.) that can occur only within a section-definition directive. This symbol refers to the current address of **ld**'s location counter. Thus, assignment expressions involving dot (.) are evaluated during the allocation phase of **ld**. Assigning a value to the dot symbol within a section-definition directive can increment (but not decrement) **ld**'s location counter and can create holes within the section, as described in "Section Definition Directives." Assigning the value of dot (.) to another symbol permits the final allocated address (of a particular point within the link edit run) to be saved.

align is provided as a shorthand notation to allow alignment of a symbol to an *n*-byte boundary within an output section, where *n* is a power of 2. For example, the expression

```
align(n)
```

is equivalent to

```
(. + n - 1) & ~ (n - 1)
```

SIZEOF and ADDR are pseudo-functions that, given the name of a section, return the size or address of the section respectively. They may be used in symbol definitions outside of section directives.

Link editor expressions may have either an absolute or a relocatable value. When **ld** creates a symbol through an assignment statement, the symbol takes on the type of the expression. That type depends on the following rules:

- An expression with a single relocatable symbol (and zero or more constants or absolute symbols) is relocatable.
- The difference of two relocatable symbols from the same section is absolute.
- All valid expressions are combinations of the above.

2.3 Specifying a Memory Configuration

The MEMORY directive is used to specify the following:

- The total size of the virtual space of the target machine
- The configured and unconfigured areas of the virtual space

The syntax of the MEMORY directive is as follows:

```
MEMORY
{
        name1 (attr) :  origin = n1, length = n2
        name2 (attr) :  origin = n3, length = n4
        etc.
}
```

The keyword **origin** (or **org** or **o**) must precede the origin of a memory range, and **length** (or **len** or **l**) must precede the length as shown in the above prototype. The **origin** operand refers to the virtual address of the memory range. **origin** and **length** are entered as long integer constants in either decimal, octal, or hexadecimal (standard C syntax). **origin** and **length** specifications, as well as individual MEMORY directives, may be separated by white space or a comma.

If no MEMORY directives are supplied, **ld** assumes that all memory is configured.

MEMORY directives assign an arbitrary name of up to eight characters to a virtual address range. Output sections can then be forced to be bound to virtual addresses within specifically named memory areas. Memory names may contain uppercase or lowercase letters, digits, and the special characters **$**, **.**, or **_**. Names of memory ranges are used by **ld** only and are not carried in the output file symbol table or headers.

When MEMORY directives are used, all virtual memory not described in a MEMORY directive is considered to be unconfigured. Unconfigured memory is not used in **ld**'s allocation process; hence nothing except DSECT sections can be link edited or bound to an address within unconfigured memory. For information on DSECT, refer to "DSECT, COPY, NOLOAD, INFO, and OVERLAY" later in this chapter.

As an option on the MEMORY directive, attributes may be associated with a named memory area. These attributes are currently accepted:

1. R : readable memory
2. W : writable memory
3. X : executable, i.e., instructions may reside in this memory
4. I : initializable, i.e., stack areas are typically not initialized

If no attributes are specified on a MEMORY directive or if no MEMORY directives are supplied, memory areas assume the attributes of R, W, X, and I.

By specifying MEMORY directives, **ld** can be told that memory is configured in some manner other than the default. (Default memory size is machine-dependent.) For example, if it is necessary to prevent anything from being linked to the first 0x10000 words of memory, a MEMORY directive can accomplish this, as follows:

```
MEMORY
{
        valid : org = 0x10000, len = 0xFE0000
}
```

2.4 Section Definition Directives

The SECTIONS directive describes how input sections are to be combined, to direct where to place output sections (both in relation to each other and to the entire virtual memory space), and to permit the renaming of output sections.

If no SECTIONS directives are given, all input sections of the same name appear in an output section of that name. If two object files are linked, one containing sections `s1` and `s2` and the other containing sections `s3` and `s4`, the output object file contains the four sections `s1`, `s2`, `s3`, and `s4`. The order of these sections depends on the order in which **ld** sees the input files.

Figure 2 shows the basic syntax of the SECTIONS directive:

```
SECTIONS
{
        secname1 :
        {
                file_specifications,
                assignment_statements
        }
        secname2 :
        {
                file_specifications,
                assignment_statements
        }
etc.
}
```

Figure 2. Syntax of the SECTIONS directive.

The types of section definition directives are discussed in the remainder of this section.

File Specifications

Within a section definition, the files and sections of files to be included in the output section are listed in the order in which they are to appear in the output section. Sections from an input file are specified by

```
filename ( secname )
```

or

```
filename ( secnam1 secnam2 . . . )
```

Sections of an input file are separated either by white space or commas as are the file specifications themselves. The following definition may be used in the same way to refer to all the uninitialized, unallocated global symbols in a file.

```
filename [COMMON]
```

If a file name appears with no sections listed, all sections from the file (but not the uninitialized, unallocated globals) are linked into the current output section. Consider the example in Figure 3.

```
SECTIONS
{
        outsec1:
        {
                file1.o (sec1)
                file2.o
                file3.o (sec1, sec2)
        }
}
```

Figure 3. Specifying output section files.

According to this directive, the order in which the input sections appear in the output section `outsec1` would be as follows:

1. Section `sec1` from file *file1.o*
2. All sections from *file2.o*, in the order they appear in the file
3. Section `sec1` from file *file3.o*, then section `sec2` from file *file3.o*

If there are any additional input files that contain input sections also named `outsec1`, these sections are linked following the last section named in the definition of `outsec1`. If there are any other input sections in *file1.o* or *file3.o*, they will be placed in output sections with the same names as the input sections unless they are included in other file specifications.

The following code can be used to indicate all previously unallocated input sections of the given name (*secname*), regardless of what input file they are contained in.

```
* (secname)
```

Load a Section at a Specified Address

Binding of an output section to a specific virtual address is accomplished by an **ld** option as shown in Figure 4.

```
SECTIONS
{
        outsec addr:
        {
                . . .
        }
        etc.
}
```

Figure 4. Loading a section in a specified address.

The *addr* is the binding address expressed as a C constant. If `outsec` does not fit at *addr* (perhaps because of holes in the memory configuration or because `outsec` is too large to fit without overlapping some other output section), **ld** issues an appropriate error message. *addr* may also be the word BIND, followed by a parenthesized expression. The expression may use the pseudo-functions SIZEOF, ADDR, or NEXT. NEXT accepts a constant and returns the first multiple of that value that falls into configured unallocated memory; SIZEOF and ADDR accept previously defined sections.

As long as output sections do not overlap and there is enough space, they can be bound anywhere in configured memory. The SECTIONS directives defining output sections need not be given to **ld** in any particular order, unless SIZEOF or ADDR is used.

ld does not ensure that each section's size consists of an even number of bytes or that each section starts on an even byte boundary. The assembler ensures that the size (in bytes) of a section is evenly divisible by 4. **ld** directives can be used to force a section to start on an odd byte boundary although this is not recommended. If a section starts on an odd byte boundary, the section's contents are either accessed incorrectly or are not executed properly. When a user specifies an odd byte boundary, **ld** issues a warning message.

Aligning an Output Section

It is possible to request that an output section be bound to a virtual address that falls on an *n*-byte boundary, where *n* is a power of 2. The ALIGN option of the SECTIONS directive performs this function, so that the option

```
ALIGN(n)
```

is equivalent to specifying a binding address of

```
( . + n - 1) & ~ (n - 1)
```

In the following example, the output section `outsec` is not bound to any given address but is placed at some virtual address that is a multiple of 0x20000 (e.g., at address 0x0, 0x20000, 0x40000, 0x60000, and so on).

```
SECTIONS
{
        outsec  ALIGN(0x20000) :
        {
                . . .
        }
        etc.
}
```

Grouping Sections Together

The default allocation algorithm for **ld**(1) performs these tasks:

1. Links all input `.init` sections together, followed by `.text` sections, into one output section. This output section is called `.text` and is bound to an address of 0x0 plus the size of all headers in the output file.
2. Links all input `.data` sections together into one output section. This output section is called `.data` and, in paging systems, is bound to an address aligned to a machine-dependent constant plus a number dependent on the size of headers and text.
3. Links all input `.bss` sections and all uninitialized, unallocated global symbols together into one output section. This output section is called `.bss` and is allocated so as to immediately follow the output section `.data`. Note that the output section `.bss` is not given any particular address alignment.

Specifying any SECTIONS directives results in this default allocation not being performed. The default allocation of **ld** is equivalent to supplying the following directive, where *align_value* is a machine dependent constant.

```
SECTIONS
{
        .text sizeof_headers : { *(.init) *(.text) }
        GROUP BIND( NEXT(align_value) +
       ((SIZEOF(.text) + ADDR(.text)) % 0x2000)) :
        {
                .data     : { }
                .bss      : { }
                .shdata     : { }
                .shbss      : { }
        }
}
```

The GROUP command ensures that the two output sections, `.data` and `.bss`, are allocated (grouped) together. Binding or alignment information is supplied only for the group and not for the output sections contained within the group. The sections making up the group are allocated in the order listed in the directive.

If `.text`, `.data`, and `.bss` are to be placed in the same segment, the SECTIONS directive is used as shown in Figure 5.

```
SECTIONS
{
        GROUP                   :
        {
                .text           : { }
                .data           : { }
                .bss            : { }
        }
}
```

Figure 5. Placing output sections in the same segment.

Note that there are still three output sections (`.text`, `.data`, and `.bss`), but now they are allocated into consecutive virtual memory.

This entire group of output sections could be bound to a starting address or aligned simply by adding a field to the GROUP directive. To bind to 0xC0000, use the following:

```
GROUP 0xC0000 : {
```

To align to 0x10000, use the following:

```
GROUP ALIGN(0x10000) : {
```

With this addition, first the output section `.text` is bound at 0xC0000 (or is aligned to 0x10000); then the remaining entries of the group are allocated in order of their appearance into the next available memory locations.

When the GROUP directive is not used, each output section is treated as an independent entity as shown in Figure 6:

```
SECTIONS
{
        .text    : { }
        .data ALIGN(0x20000)   : { }
        .bss     : { }
}
```

Figure 6. Output sections placed in separate segments.

The .text section starts at virtual address 0x0 (if it is in configured memory) and the .data section at a virtual address aligned to 0x20000. The .bss section follows immediately after the .text section if there is enough space. If there is not, it follows the .data section. The order in which output sections are defined to ld cannot be used to force a certain allocation order in the output file.

Creating Holes Within Output Sections

The special dot (.) symbol appears only within section definitions and assignment statements. When it appears on the left side of an assignment statement, dot (.) causes ld's location counter to be incremented or reset and a hole left in the output section. Holes built into output sections in this manner take up physical space in the output file and are initialized using a fill character (either the default fill character (0x00) or a supplied fill character). See the definition of the –f option in "Using the Link Editor" and the discussion of filling holes in "Initialized Section Holes or .bss Sections" in this chapter.

Consider the section definition in Figure 7.

```
outsec:
{
        . += 0x1000;
        f1.o (.text)
        . += 0x100;
        f2.o (.text)
        . = align (4);
        f3.o (.text)
}
```

Figure 7. Creating holes within output sections.

The effect of this command is as follows:

1. A 0x1000 byte hole, filled with the default fill character, is left at the beginning of the section. Input section .text of input file f1.o is linked after this hole.
2. The .text section of input file f2.o begins at 0x100 bytes following the end of f1.o (.text).

3. The `.text` section of `f3.o` is linked to start at the next full word boundary following the `.text` section of `f2.o` with respect to the beginning of `outsec`.

For the purposes of allocating and aligning addresses within an output section, **ld** treats the output section as if it began at address zero. As a result, if, in the above example, `outsec` ultimately is linked to start at an odd address, then the part of `outsec` built from `f3.o (.text)` also starts at an odd address—even though `f3.o (.text)` is aligned to a full word boundary. This is prevented by specifying an alignment factor for the entire output section.

```
outsec ALIGN(4) : {
```

Note that the assembler **as**, as well as the compiler, always pads the sections it generates to a full word length making explicit alignment specifications unnecessary.

Expressions that decrement dot (.) are illegal. For example, subtracting a value from the location counter is not allowed because overwrites are not allowed. The most common operators in expressions that assign a value to . are **+=** and **align**.

Creating and Defining Symbols at Link-Edit Time

ld assignment statements can be used to create and define symbols at link time. These statements can also be used to changed the address of a symbol created by the assembler or the compiler. Typically, there are three types of assignments:

1. Use of dot (.) to adjust **ld**'s location counter during allocation.
2. Use of dot (.) to assign an allocation-dependent value to a symbol.
3. Assigning an allocation-independent value to a symbol.

Case 1 was discussed in the previous section.

Case 2 provides a means to assign addresses (known only after allocation) to symbols. Figure 8 shows an example of this type of assignment. The symbol **s2_start** is defined to be the address of *file2.o(s2)*, and **s2_end** is the address of the last byte of *file2.o(s2)*.

```
SECTIONS
{
        outsc1: {...}
        outsc2:
        {
                file1.o (s1)
                s2_start = . ;
                file2.o (s2)
                s2_end = . - 1;
        }
}
```

Figure 8. Assigning an allocation-dependent value to a symbol.

Figure 9 shows an example of assigning an allocation-independent value to a symbol. The symbol **mark** is created and is equal to the address of the first byte beyond the end of *file1.o*'s `.data` section. Four bytes are reserved for a future run-time initialization of the symbol **mark**. The type of the symbol is a long integer (32 bits).

```
SECTIONS
{
        outsc1:
        {
                file1.o (.data)
                mark = .;
                . += 4;
                file2.o (.data)
        }
}
```

Figure 9. Assigning an allocation-independent value to a symbol.

Assignment instructions involving dot (.) must appear within SECTIONS definitions because they are evaluated during allocation. Assignment instructions that do not involve dot (.) can appear within SECTIONS definitions but typically do not. Such instructions are evaluated after allocation is complete. Reassignment of a defined symbol to a different address is dangerous. For example, if a symbol within `.data` is defined, initialized, and referenced within a set of object files being link-edited, the symbol table entry for that symbol is changed to reflect the new, reassigned physical address. However, the associated initialized data is not moved to the new address, and there may be references to the old address. **ld** issues warning messages for each defined symbol that is being redefined within an ifile. However, assignments of absolute values to new symbols are safe because there are no references or initialized data associated with the symbol.

Allocating a Section into Named Memory

A section can be linked (somewhere) within a specifically named memory named in a MEMORY directive. (The > notation is borrowed from the operating system concept of redirected output.)

The sample code in Figure 10 directs **ld** to place `outsec1` anywhere within the memory area named `mem1` (that is, somewhere within the address range 0x0-0xFFFF or 0x120000-0x123FFF). The `outsec2` is to be placed somewhere in the address range 0x70000-0xAFFFF.

```
MEMORY
{
        mem1:           o=0x000000    l=0x10000
        mem2 (RW):      o=0x020000    l=0x40000
        mem3 (RW):      o=0x070000    l=0x40000
        mem1:           o=0x120000    l=0x04000
}

SECTIONS
{
        outsec1: { f1.o(.data) } > mem1
        outsec2: { f2.o(.data) } > mem3
}
```

Figure 10. Allocating a section into named memory.

Initialized Section Holes or .bss Sections

When holes are created within a section, as in the example in "Creating Holes within Output Sections", **ld** normally puts out bytes of zero as fill. By default, `.bss` sections are not initialized at all; that is, no initialized data is generated for any `.bss` section by the assembler nor supplied by the link editor, not even zeros.

Initialization options can be used in a SECTIONS directive to set such holes or output `.bss` sections to an arbitrary 2-byte pattern. Such initialization options apply only to `.bss` sections or holes. As an example, an application might want an uninitialized data table to be initialized to a constant value without recompiling the *.o* file or a hole in the text area to be filled with a transfer to an error routine.

Either specific areas within an output section or the entire output section may be specified as being initialized. However, because no text is generated for an uninitialized `.bss` section, if part of such a section is initialized, then the entire section is initialized. In other words, if a `.bss` section is to be combined with a `.text` or `.data` section (both of which are initialized) or if part of an output `.bss` section is to be initialized, then one of the following will hold:

1. Explicit initialization options must be used to initialize all `.bss` sections in the output section.
2. **ld** uses the default fill value to initialize all `.bss` sections in the output section.

Figure 11 shows an example of initializing section holes. The 0x200 byte hole in section `sec1` is filled with the value 0xDFFF. In section `sec2`, `f1.o(.bss)` is initialized to the default fill value of 0x00, and `f2.o(.bss)` is initialized to 0x1234. All `.bss` sections within `sec3` as well as all holes are initialized to 0xFFFF. Section `sec4` is not initialized; that is, no data is written to the object file for this section.

```
SECTIONS
{
        sec1:
        {
                f1.o
                . =+ 0x200;
                f2.o (.text)
        } = 0xDFFF
        sec2:
        {
                f1.o (.bss)
                f2.o (.bss) = 0x1234
        }
        sec3:
        {
                f3.o (.bss)
                . . .
        } = 0xFFFF
         sec4: { f4.o (.bss) }
}
```

Figure 11. Initializing section holes.

3 Notes and Special Considerations

The following sections address special cases of the linking process.

3.1 Use of Archive Libraries

Each entry of an archive library (for example, **libc.a**) is a complete object file. Archive libraries are created with the **ar**(1) command from object files generated by **cc** or **as**. **ld** processes archive libraries using selective inclusion; only those entries that resolve existing undefined-symbol references are taken from the library for link editing. Libraries can be placed both inside and outside section definitions. In both cases, a library entry is included for linking whenever the following conditions occur:

1. A reference to a symbol defined in that entry exists.
2. The reference is found by **ld** prior to the actual scanning of the library.

When a library entry is included by searching the library inside a SECTIONS directive, all input sections from the library entry are included in the output section being defined. When a library entry is included by searching the library outside of a SECTIONS directive, all input sections from the library entry are included into the output section with the same name. If necessary, new output sections are defined to provide a place to put the input sections. Note, however, that:

1. Specific entries of a library cannot be referenced explicitly in an ifile.
2. The default rules for the placement of entries and sections cannot be overridden when they apply to archive library entries.

The **ld –l <file name>** option is a shorthand notation for specifying an input file coming from a predefined set of directories and having a predefined name. By convention, such files are archive libraries. However, they need not be so. Furthermore, archive libraries can be specified without using the –l option by simply giving the full or relative pathname.

The order of archive libraries is important on the command line because an entry must satisfy a reference that is known to be unresolved at the time the library is searched before it can be extracted from the library. Archive libraries can be specified more than once. They are searched every time they are encountered. Archive files have a symbol table at the beginning of the archive. **ld** will cycle through this symbol table until it has determined that it cannot resolve any more references from that library.

Consider the following example:

1. The input files *file1.o* and *file2.o* each contain a reference to the external function FCN.
2. Input *file1.o* contains a reference to symbol ABC.
3. Input *file2.o* contains a reference to symbol XYZ.
4. Library **liba.a**, entry 0, contains a definition of XYZ.
5. Library **libc.a**, entry 0, contains a definition of ABC.
6. Both libraries have an entry 1 that defines FCN.

If the **ld** command were entered as follows, then the FCN references are satisfied by **liba.a**, entry 1, ABC is obtained from **libc.a**, entry 0, and XYZ remains undefined (because the library **liba.a** is searched before *file2.o* is specified).

```
$ ld file1.o -la file2.o -lc
```

If the **ld** command were entered as follows, then the FCN references are satisfied by **liba.a**, entry 1, ABC is obtained from **libc.a**, entry 0, and XYZ is obtained from **liba.a**, entry 0.

```
$ ld file1.o file2.o -la -lc
```

If the **ld** command were entered as follows, then the FCN references is satisfied by **libc.a**, entry 1, ABC is obtained from **libc.a**, entry 0, and XYZ is obtained from **liba.a**, entry 0.

```
ld file1.o file2.o -lc -la
```

The **-u** option is used to force the linking of library entries when the link edit run does not contain an actual external reference to the entries. For example, the following command creates an undefined symbol called **rout1** in **ld**'s global symbol table.

```
$ l -u rout1 -la
```

If any entry of library **liba.a** defines this symbol, it (and perhaps other entries as well) is extracted. Without the **-u** option, there would have been no unresolved references or undefined symbols to cause **ld** to search the archive library.

3.2 Nonrelocatable Input Files

If a file produced by **ld** is intended to be used in a subsequent **ld** run, the first **ld** run should have the **–r** option set. This preserves relocation information and permits the sections of the file to be relocated by the subsequent run.

If an input file to **ld** does not have relocation or symbol table information (perhaps from the action of a **strip**(1) command, or from being link edited without a **–r** option or with a **–s** option), the link edit run continues using the nonrelocatable input file.

For such a link edit to be successful (that is, to actually and correctly link edit all input files, relocate all symbols, resolve unresolved references, and so on), two conditions on the nonrelocatable input files must be met.

1. Each input file must have no unresolved external references.
2. Each input file must be bound to the exact same virtual address as it was bound to in the **ld** run that created it.

NOTE

If these two conditions are not met for all nonrelocatable input files, no error messages are issued. Because of this fact, extreme care must be taken when supplying such input files to **ld**.

3.3 Changing the Entry Point

The optional header in **a.out** contains a field for the (primary) entry point of the file. This field is set using one of the following rules which are listed in the order they are applied:

1. The value of the symbol specified with the **–e** option, if present, is used.
2. The value of the symbol **_start**, if present, is used.
3. The value of the symbol **main**, if present, is used.
4. The value zero is used.

Thus, an explicit entry point can be assigned to this **a.out** header field through the **-e** option or by using an assignment instruction in an ifile of this form:

```
_start    =    expression;
```

If **ld** is called through **cc**(1), a startup routine is automatically linked in. Then, when the program is executed, the routine **exit**(1) is called after the main routine finishes to close file descriptors and do other cleanup. The user must therefore be careful when calling **ld** directly or when changing the entry point. The user must supply the startup routine or make sure that the program always calls exit rather than falling through the end. Otherwise, the program will dump core.

3.4 Dealing With Holes in Physical Memory

When memory configurations are defined such that unconfigured areas exist in the virtual memory, each application or user must assume the responsibility of forming output sections that will fit into memory. For example, assume that memory is configured as follows:

```
MEMORY
{
        mem1:       o = 0x00000       l = 0x02000
        mem2:       o = 0x40000       l = 0x05000
        mem3:       o = 0x20000       l = 0x10000
}
```

Let the files *f1.o, f2.o, . . . fn.o* each contain three sections `.text`, `.data`, and `.bss`, and suppose the combined `.text` section is 0x12000 bytes. There is no configured area of memory in which this section can be placed. Appropriate directives must be supplied to break up the `.text` output section so **ld** may do allocation. Figure 12 shows an example of this.

```
SECTIONS
{
        txt1:
        {
                f1.o (.text)
                f2.o (.text)
                f3.o (.text)
        }
        txt2:
        {
                f4.o (.text)
                f5.o (.text)
                f6.o (.text)
        }
        etc.
}
```

Figure 12. Dealing with holes in physical memory.

3.5 Allocation Algorithm

An output section is formed either as a result of a SECTIONS directive, by combining input sections of the same name, or by combining `.text` and `.init` into `.text`. An output section can have zero or more input sections comprising it. After the composition of an output section is determined, it must then be allocated into configured virtual memory. **ld** uses an algorithm that attempts to minimize fragmentation of memory, and hence increases the possibility that a link edit run will be able to allocate all output sections within the specified virtual memory configuration. The algorithm proceeds as follows:

1. Any output sections for which explicit binding addresses were specified are allocated.
2. Any output sections to be included in a specific named memory are allocated. In both this and the succeeding step, each output section is placed into the first available space within the (named) memory with any alignment taken into consideration.

3. Output sections not handled by one of the above steps are allocated.

If all memory is contiguous and configured (the default case), and no SECTIONS directives are given, then output sections are allocated in the order they appear to **ld**. Otherwise, output sections are allocated in the order they were defined or made known to **ld** into the first available space they fit.

3.6 Incremental Link Editing

As previously mentioned, the output of **ld** can be used as an input file to subsequent **ld** runs, providing that the relocation information is retained (**-r** option). Large applications may find it desirable to partition their C programs into subsystems, link each subsystem independently, and then link edit the entire application. Figure 13 shows the steps to create and link subsystems.

```
Step 1:  ld -r -o outfile1 ifile1 infile1.o
        /* ifile1  */
        SECTIONS
        {
                ss1:
                {
                        f1.o
                        f2.o
                        . . .
                        fn.o
                }
        }

Step 2:  ld -r -o outfile2 ifile2 infile2.o

        /* ifile2  */
        SECTIONS
        {
                ss2:
                {
                        g1.o
                        g2.o
                        . . .
                        gn.o
                }
        }

Step 3:  ld -a -o final.out outfile1 outfile2
```

Figure 13. Creating and linking subsystems.

By judiciously forming subsystems, you can create applications in which it is necessary to relink only a portion of the total link edit when a few files are recompiled.

To apply this technique, there are two simple rules:

1. Intermediate link edits should contain only SECTIONS declarations and be concerned only with the formation of output sections from input files and input sections. No binding of output sections should be done in these runs.

2. All allocation and memory directives, as well as any assignment statements, are included only in the final **ld** call.

3.7 DSECT, COPY, NOLOAD, INFO, and OVERLAY Sections

Sections may be given a type in a section definition as shown in the following example:

```
SECTIONS
{
        name1 0x200000 (DSECT)      : { file1.o }
        name2 0x400000 (COPY)       : { file2.o }
        name3 0x600000 (NOLOAD)     : { file3.o }
        name4          (INFO)       : { file4.o }
        name5 0x900000 (OVERLAY)    : { file5.o }

}
```

The DSECT option creates what is called a dummy section. A dummy section has the following properties:

1. It does not participate in the memory allocation for output sections. As a result, it takes up no memory and does not show up in the memory map generated by **ld**.
2. It may overlay other output sections and even unconfigured memory. DSECTs may overlay other DSECTs.
3. The global symbols defined within the dummy section are relocated normally. That is, they appear in the output file's symbol table with the same value they would have had if the DSECT were actually loaded at its virtual address. DSECT-defined symbols may be referenced by other input sections. Undefined external symbols found within a DSECT cause specified archive libraries to be searched and any entries which define such symbols are link edited normally (that is, not as a DSECT).
4. None of the section contents, relocation information, or line number information associated with the section is written to the output file.

In the above example, none of the sections from *file1.o* are allocated, but all symbols are relocated as though the sections were link edited at the specified address. Other sections could refer to any of the global symbols and they are resolved correctly.

A copy section created by the COPY option is similar to a dummy section. The only difference between a copy section and a dummy section is that the contents of a copy section and all associated information is written to the output file.

An INFO section is the same as a COPY section but its purpose is to carry information about the object file whereas the COPY section may contain valid text and data. INFO sections are usually used to contain file version identification information.

A section with the type of NOLOAD differs in only one respect from a normal output section: its text and/or data is not written to the output file. A NOLOAD section is allocated virtual space, appears in the memory map, and so on.

An OVERLAY section is relocated and written to the output file. It is different from a normal section in that it is not allocated and may overlay other sections or unconfigured memory.

3.8 Output File Blocking

The BLOCK option (applied to any output section or GROUP directive) is used to direct **ld** to align a section at a specified byte offset in the output file. It has no effect on the address at which the section is allocated nor on any part of the link edit process. It is used purely to adjust the physical position of the section in the output file.

With the following SECTIONS directive, **ld** assures that each section, `.text` and `.data`, is physically written at a file offset, which is a multiple of 0x200 (e.g., at an offset of 0, 0x200, 0x400, and so forth, in the file).

```
SECTIONS
{
        .text BLOCK(0x200) : { }
        .data ALIGN(0x20000) BLOCK(0x200) : { }
}
```

4 Syntax Diagram for Input Directives

The table below lists the directives and describes their syntax. Square brackets and curly braces do double duty in this diagram.

Where the actual symbols, [] and {} are used, they are part of the syntax and must be present when the directive is specified.

Where you see the brackets **[** and **]** larger and in bold, it means the material enclosed is optional.

Where you see the braces **{** and **}** larger and in bold, it means multiple occurrences of the material enclosed are permitted.

Table 1
Syntax Diagram for Input Directives

Directives	Expanded Directives
<ifile>	{<cmd>}
<cmd>	<memory> <sections> <assignment> <filename> <flags>
<memory>	MEMORY { <memory_spec> { [,] <memory_spec> } }
<memory_spec>	<name> [<attributes>] : <origin_spec> [,] <length_spec>
<attributes>	({ R | W | X | I })
<origin_spec>	<origin> = <long>
<lenth_spec>	<length> = <long>
<origin>	ORIGIN | o | org | origin
<length>	LENGTH | l | len | length

Table 1
Syntax Diagram for Input Directives (cont.)

Directives	Expanded Directives
<sections>	SECTIONS { {<sec_or_group>} }
<sec_or_group>	<section> | <group> | <library>
<group>	GROUP <group_options> : { <section_list> } [<mem_spec>]
<section_list>	<section> { [,] <section> }
<section>	<name> <sec_options> : { <statement> } [<fill>] [<mem_spec>]
<group_options>	[<addr>] | [<align_option>] [<block_option>]
<sec_options>	[<addr>] | [<align_option>] [<block_option>] [<type_option>]
<addr>	<long> | <bind>(<expr>)
<alignoption>	<align> (<expr>)
<align>	ALIGN | align
<block_option>	<block> (<long>)
<block>	BLOCK | block
<type_option>	(DSECT) | (NOLOAD) | (COPY) | (INFO) | (OVERLAY)
<fill>	= <long>
<mem_spec>	> <name> > <attributes>

Table 1
Syntax Diagram for Input Directives (cont.)

Directives	Expanded Directives
<statement>	<filename> <filename> (<name_list>) | [COMMON] * (<name_list>) | [COMMON] <assignment> <library> *null*
<name_list>	<section_name> [,] { <section_name> }
<library>	–l<name>
<bind>	BIND | bind
<assignment>	<lside> <assign_op> <expr> <end>
<lside>	<name> | .
<assign_op>	= | += | –= | *= |/ \\=
<end>	; | ,
<expr>	<expr> <binary_op> <expr> <term>
<binary_op>	* | / | % + | – >> | << == | != | > | < | <= | >= & | && ||

Table 1
Syntax Diagram for Input Directives (cont.)

Directives	Expanded Directives
<term>	<long> <name> <align> (<term>) (<expr>) <unary_op> <term> <phy> (<lside>) <sizeof>(<sectionname>) <next>(<long>) <addr>(<sectionname>)
<unary_op>	! | –
<phy>	PHY | phy
<sizeof>	SIZEOF | sizeof
<next>	NEXT | next
<addr>	ADDR | addr
<name>	Any valid symbol name
<long>	Any valid long integer constant
<wht_space>	Blanks, tabs, and newlines
<filename>	Any valid operating system filename. This may include a full or partial pathname.
<sectionname>	Any valid section name, up to 8 characters
<path_name>	Any valid operating system pathname (full or partial)

Index

A

ADDR pseudo-function 7,12
Address space 3
Aligning a symbol to a byte-boundary 7
Aligning an output section 13
Allocating sections into named memory 19
Allocation algorithm 27
Archive libraries 23
Assignment statements 6

B

Binding output sections 3
Blocking output files 31
`bss` sections 13
Byte-boundary symbol alignment 7

C

Changing entry points 25
Configuring memory 2
COPY sections 31
Creating holes in output sections 16
Creating symbols at link-time 17

D

`data` sections 13
Defining symbols at link-time 17
Directives
  GROUP 14
  MEMORY 8,9,19
  SECTIONS 9,10,12,13,19,23
  syntax 32
Dot symbol 7,16,19
DSECT section 30

E

Entry points 25
Expressions 4,7
  absolute value 7
  relocatable value 7

F

Files
  blocking 31
  `bss` sections 13
  `data` sections 4,13
  ifile 2
  `init` sections 13
  object 3
  relocatable 3,25
  section definitions 10
  specifications 10
  `text` sections 4

G-H

GROUP directive 14
Grouping sections 13
Holes in output sections 16
Holes in physical memory 26

I

Ifile 2
Incremental link editing 28,30
INFO sections 31
`init` sections 13
Initialization options 20
Initialized `.bss` sections 20
Initialized section holes 20
Input file relocation 25

J-K-L

ld
command 2
invoking 2
lex, used by **ld** 5
Libraries, archive 23
Load section at specified address 12

M

Memory configuration 2,8
holes in physical memory 26
Memory configuration algorithm 27
MEMORY directive 8,19

N

Names, reserved 5
NEXT pseudo-function 12
NOLOAD sections 31
Nonrelocatable input files 25

O

Object file section 3
Object files 3
Operator precedence 5
Output file blocking 31
OVERLAY sections 31
Overview 1

P-Q-R

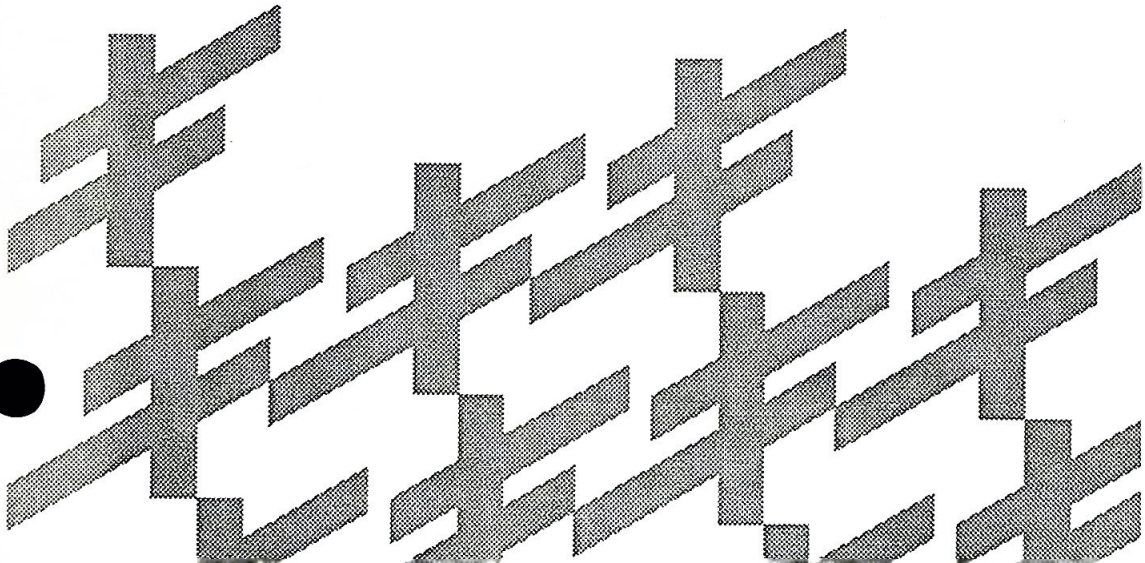
Physical address 3
Precedence of operators 5
Relocating input file 25
Reserved names 5
Running **ld** 2

S-T-U-V-W-X-Y-Z

Section definitions 10
Sections 3
  aligning output 13
  `bss` 13
  COPY 31
  `data` 4,13
  DSECT 30
  grouping together 13
  holes 3
  INFO 31
  `init` 13
  load at specified address 12
  NOLOAD 31
  OVERLAY 31
  relocation 3
  `text` 4
SECTIONS directive 9,12,13,19,23
  file specifications 10
  initialization options 20
Sections, allocating 19
SIZEOF pseudo-function 7,12
Specifying archive libraries 23
Specifying memory configuration 8
Specifying starting addresses 3
Syntax
  input directives 32
Text sections 4
Virtual memory 2

DYNIX/ptx®
Assembly Language
User's Manual

1003-48546-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

Symmetry is a trademark of Sequent Computer Systems, Inc.

Copyright 1988, 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America

Contents

Chapter 1. Overview
1.1 Introduction 1-1
1.2 Running the Assembler 1-1
1.3 File Formats 1-2
1.4 Statement Formats 1-4
1.5 Expressions 1-8

Chapter 2. Assembler Directives
2.1 Introduction 2-1

Chapter 3. 80386 Instruction Summary
3.1 Introduction 3-1
3.2 Notational Conventions 3-1
3.3 Operands 3-2
3.4 Condition Codes 3-6
3.5 Classified List of Instructions 3-8
3.6 Differing Mnemonics 3-14
3.7 80386 Instructions 3-15

Chapter 4. 80387 FPU Instruction Summary
4.1 Introduction 4-1
4.2 Notational Conventions 4-2
4.3 Operands 4-2
4.4 Classified List of Instructions 4-4
4.5 80387 FPU Instructions 4-8

Chapter 5. Programming Example
5.1 Introduction 5-1
5.2 Factorial Numbers 5-1

Appendix A. Porting Considerations
A.1 Introduction A-1
A.2 Porting a System V i386 Program to DYNIX/ptx A-1
A.3 Porting an i386 Program from DYNIX 3.0 to DYNIX/ptx A-1

Appendix B. 1167Floating-point Accelerator Operation Summary
B.1 Introduction B-1
B.2 Notational Conventions B-1
B.3 Operands B-2
B.4 1167 Floating-point Accelerator Operations B-3

Figures

1-1 Sample assembler program using numeric labels 1-6

3-1 Sample valid expressions and addressing modes 3-5
3-2 Sample invalid expressions 3-6

5-1 A factorial routine with a C interface 5-2

Tables

1-1 Expression Operators 1-13

2-1 Assembler Directives 2-1

3-1 80386 Registers 3-3
3-2 Addressing Modes 3-4
3-3 Condition Codes 3-6
3-4 Differing Mnemonics 3-14

4-1 Instruction Formats 4-4

About This Manual

Overview

This manual describes the Symmetry assembler that runs under the DYNIX/ptx operating system.

Assumptions about the Reader

This manual is intended for assembly language programmers and compiler writers. We assume that the reader is familiar with assembly language programming, Common Object File Format (COFF), and the Intel 80386 processor architecture.

Organization

This manual is organized as follows:

Assembler invocation and assembly language syntax are described in Chapter 1.

Assembler directives, their syntax and use are described in Chapter 2.

80386 CPU instructions are described in Chapter 3.

80387 CPU instructions are described in Chapter 4.

A sample assembly language source file that illustrates principles of Symmetry assembly language programming, the directives and statement syntax accepted by the assembler, and the subroutine call interface between C and assembly language routines appears in Chapter 5.

Porting considerations for porting i386 programs are described in Appendix A:

Macros patterned after Weitek 1167 floating-point accelerator instructions are described in Appendix B.

Notational Conventions

The following notational conventions are used in this manual:

italic	Italicized words indicate that you must supply an item of the indicated type.
`constant width`	Words shown in constant-width font indicate that you must type the item exactly as shown.
`[]`	Brackets indicate an optional item.

Chapter 1
Overview

1.1 Introduction 1-1

1.2 Running the Assembler 1-1

1.3 File Formats 1-2
Input 1-3
Output 1-3
Sections 1-3
User-Specified Sections 1-4
Location Counter 1-4

1.4 Statement Formats 1-4
Comment Statement 1-7
Directive Statement 1-7
Instruction Statement 1-8

1.5 Expressions 1-8
Operands 1-9
Undefined 1-9
Undefined External 1-9
Absolute 1-9
Absolute External 1-9
Text (Relocatable) 1-9
Text External 1-10
Data (Relocatable) 1-10
Data External 1-10
Shared Data (Relocatable) 1-10
Shared Data External 1-10
bss (Relocatable) 1-10
bss External 1-10
Shared bss (Relocatable) 1-11
Shared bss External 1-11
Constants 1-11
Operators 1-13
Evaluating Expressions 1-13
Combining Types in Expressions 1-13

Figures

1-1 Sample assembler program using numeric labels 1-6

Tables

1-1 Expression Operators 1-13

Chapter 1
Overview

1.1 Introduction

This chapter describes the invocation, file and statement formats, and expression usage of the assembler, **as**. This assembler is designed to assemble the code produced by the C compiler; however, certain features have been added to handle user-written source code.

1.2 Running the Assembler

To invoke the assembler, use the **as** command. The **as** command has the following syntax:

as [*options*] [*file*]

options	Assembler options from the following list
file	An assembly language source file to be assembled

The assembler accepts the following options:

-dl	Omits line number information from the output file
-k	Assembles the source file for stand-alone execution. This option is not for use with user programs.
-m	Runs the **m4** macro processor on the input file before assembling it
-n	Turns off branch optimization in which the assembler selects the short form of branch instructions whenever possible. By default, branch optimization takes place.

–o *outfile*	Puts the assembly output in *outfile*
–r	Makes data read-only by merging the .data section with the .text section. For more information, refer to Section 1.3.
–R	Removes (unlinks) the source file after assembly
–u	Removes unreferenced debugging symbols from the symbol table
–V	Writes the assembler version number to standard error
–Y [md],*dir*	Searches *dir*, rather than the default pathname, for the **m4** preprocessor (if you specify the m qualifier) and the file of predefined macros (if you specify the d qualifier). You can specify either one qualifier or both of them.

Only one source file can be specified per command line. The assembler reads the assembly language program in the source file and writes the object module to an output file. If you do not specify a source file, the assembler takes its input from the standard input and places its output in the a.out file.

You can specify the name of the output file with the **–o** option. If you do, the assembler will append a *.o* suffix to the name you specify. If you do not specify a name, the assembler uses the input file name to name the output file according to the following scheme:

- If the input file name ends with *.s*, the assembler replaces *.s* with *.o*
- If the input file name does not end with *.s*, the assembler appends *.o* to the name.

1.3 File Formats

The assembler accepts a text file as input and produces an object file in the Common Object File Format (COFF). The following text describes the input file format and the use of sections in the object file.

1.3.1 Input

An assembly language source file consists of a sequence of lines, each of which contains an arbitrary number of characters and ends with a newline character (ASCII LF). Each line contains one or more of these types of statements: comment, directive, or instruction. If multiple statements appear on a line, each statement must be separated from the next by a semicolon (;).

Each type of statement can be modified with either a label or a comment. Statements, along with labels and comments, are discussed more fully in section 1.4.

1.3.2 Output

The nonexecutable object file produced by the assembler is a COFF file. This file contains relocatable machine code, initialized program data, and uninitialized program data. Each of these types of program code is placed in a specific section of the COFF file. When the object file is linked and executed, the contents of these sections are placed into memory according to the attributes of the section type. For more information on COFF files, refer to the operating system programming guide.

To prepare the object file for execution, you must run the link editor **ld** on this file. For more information on using **ld**, see the **ld** description in the language tools documentation.

Sections

The assembler divides object code into four memory sections: .comment, .text, .data, and .shdata (shared data). The .comment section contains copyright notices and software administrative information. The .text section contains the assembled program; this section is write-protected. The .data and .shdata sections contain both initialized and uninitialized data; these sections are readable and writable. (The uninitialized portions of the .data and .shdata sections are called *.bss* and *.shbss*, respectively.) The .data section is private: it cannot be shared between processes. The .shdata section can be shared among multiple processes.

Each of the three memory sections, .text, .data, and .shdata is divided into four subsections. These subsections are identified by number (for example, `.text 0` through `.text 3`). At the end of assembly, the assembler zero-fills each subsection so that the last address is a multiple of four bytes and then concatenates the subsections for each section.

You can switch between subsections with the `.text`, `.data`, and `.shdata` assembler directives. You can also allocate memory space in .bss or .shbss with the `.comm` and `.lcomm` directives. For more information on these directives, refer to Chapter 2.

User-Specified Sections

You can create and use user-specified sections with the `.section` assembler directive. A user-specified section can contain up to five subsections. For information on how to use the `.section` directive, refer to Chapter 2.

Using user-specified sections requires care. Because the `exec` program expects only one .text, one .data, and one .bss section, you must supply the code to handle the access and execution of any sections other than standard .text, .data, and .bss.

The DYNIX/ptx C and DYNIX/ptx FORTRAN compilers and the parallel programming library (*libpps*) implement shared memory with the user-specified sections .shdata and .shbss. Because *libpps* performs special processing to access .shdata and .shbss, always use *libpps* when you use .shdata and .shbss.

Location Counter

Associated with each section or subsection is a location counter which begins at zero and is incremented by 1 for each byte assembled into the section. The value of the location counter is specified by a period (.). Note that the location-counter values of subsections other than `.text 0`, `.data 0`, and `.shdata 0` are lost once the subsections are concatenated into one memory section.

1.4 Statement Formats

A statement contains from one to four fields, depending on the type of statement and the instruction to be coded. Statement fields are the label, operation, operand, and comment fields.

The label or operand field can contain a symbol. A *symbol*, sometimes called an identifier or name, consists of a sequence of letters, digits, periods, underscores, percent signs, or dollar signs. The first character must not be a digit. Symbols may be arbitrarily long (over 1000 characters); all characters are significant.

A single period (.) is a special symbol that stands for the value of the current location counter.

A single percent sign (%) by itself is not a valid symbol: it is interpreted as the modulo operator. Refer to "Operators", later in this chapter, for more information.

The assembler treats uppercase and lowercase letters as distinct characters. Assembler directives, register names, and instruction op-codes must be lowercase.

The assembler does not restrict user-defined symbols. Because the assembler distinguishes standard symbols by their position in the statement, there are no reserved words.

The following symbols are predefined as register names:

```
%eax %ebx %ecx %edx %ebp %esp %edi %esi

%ax %bx %cx %dx %bp %di %si %spi

%al %ah %bh %bl %dh %dl %ch %cl

%cs %ds %es %fs %gs %ss

%cr0 %cr2 %cr3

%db0 %db1 %db2 %db3 %db6 %db7

%tr6 %tr7

%st  %st(0) ... %st(7)   [Intel 80387 only]

%fp1 ... %fp31   [Weitek 1167 only]
```

When writing a source program, you can use three kinds of statements:

- Comment
- Assembler directive
- Assembler instruction

Note that there is no assignment statement. You should use the `.set` directive to make assignments. Refer to Chapter 2 for more information on the `.set` directive.

Separate statements from one another with newline characters or semicolons.

You can precede each type of statement with one label. If you include a label, it can appear either on the same line as the statement or on the line preceding the statement. Labels can be either named or numeric. A *named label* is any unique, valid symbol. A *numeric label* is any number from 0 to 9.

The similarity between named and numeric labels lies in their function and syntax. Named labels assign the current value and type of the location counter to the temporary name symbol; numeric labels make this assignment to the temporary number symbol. Both are followed by a colon, and both are referenced by the name or number alone (without the colon).

However, with numeric labels you can use the same number more than once in the same assembly; named labels cannot be redefined. Thus, you can use numeric labels to code repetitive patterns that occur frequently in a program. The assembler references a numeric label appearing in a program as *n* (where *n* is the number portion of the label) followed by either `f` (forward) or `b` (backward). If the label is followed by *f*, execution jumps to the next occurrence of that numeric label; if it is followed by *b*, execution jumps to the previous occurrence.

Figure 1-1 shows an assembler program that demonstrates the use of numeric labels. This program compares two values and loads the smaller value into the %edx register.

```
        .file "smaller"
        .globl main

main:   movb $5, %eax
        movb $7, %ecx
        cmpl %eax, %ecx
        jg   1f
        movl %eax, %edx
        jmp  2f

1:      movl %ecx, %edx
2:      ret
```

Figure 1-1. Sample assembler program using numeric labels.

You can freely use blanks and tabs between statement fields to improve the readability of the source program. However, blanks cannot be used within a statement field (except in a string constant). A blank or tab is required to separate adjacent symbols or constants from each other.

1.4.1 Comment Statement

You can include comments in the source program by using the slash character (/) at the beginning of each line. The assembler ignores everything that appears from the slash to the end of the line.

1.4.2 Directive Statement

Directive statements control the operation of the assembler. Directive statements have the following format:

```
[label:] directive [arg [,arg]]  [/comment]
```

label — A user-defined symbol or numeric value that provides an entry point for a subroutine or a loop. This field is optional. A label can start in any column as long as it is the first field in the instruction. For more information, refer to section 1.4.

directive — An assembler directive. The syntax and function of each assembler directive is explained more fully in Chapter 2.

arg — An argument to the assembler directive, if the directive takes arguments. See Chapter 2 for an explanation of directive arguments.

comment — A partial line of comment to document statement use. The assembler ignores everything that appears from the slash to the end of the line. This field is optional.

1.4.3 Instruction Statement

An instruction statement has the following format:

```
[label:] op-code  [operand [,operand]]  [/comment]
```

label — A user-defined symbol or numeric value that provides an entry point for a subroutine or a loop. This field is optional. A label can start in any column as long as it is the first field in the instruction. For more information, refer to section 1.4.

op-code — One of the instruction op-codes listed in Chapters 3 and 4 and Appendix B. This field is required.

operand — The value to be operated on. Different operations have different numbers of operands, and the type of each operand varies with the type of operation to be performed. For more information on this field, see the appropriate op-code description in Chapters 3 or 4 or Appendix B.

comment — A partial line of comment to document statement use. The assembler ignores everything that appears from the slash to the end of the line. This field is optional.

1.5 Expressions

An expression is a sequence of symbols representing a value. It is composed of operands, operators, and square brackets ([ ]). Examples of valid expressions are $4 + x$ and $x + [y - z] * 4$.

Each expression has one of these types: undefined, absolute, and relocatable. Each of these types can also be external. The type of the expression depends on the types of its operands.

1.5.1 Operands

Operands are the elements to be operated on. An operand consists of symbols, constants, and expressions. An example of a symbol is *x*. A constant is a value such as *232* or *0xf7*. Expressions are shown in Section 1.5.

Understanding operand types is useful when you are determining how to combine operands. The type of an operand is undefined, absolute, or relocatable. If an operand is relocatable, the value of the operand is related to its memory section. Any of these types can also be external. The following sections, beginning with "Undefined", explain how each type is defined. Section 1.5.4 contains the rules for combining operands.

Undefined

A symbol is undefined when the assembler first encounters it or it is assigned to an undefined expression.

Undefined External

A symbol that is declared with the `.globl` directive is an undefined external. Undefined externals are resolved in the link phase by **ld**, provided the symbol is defined in one of the object files that is input to **ld**.

Absolute

An absolute symbol is defined from a constant. Its value is unaffected by any possible future applications of **ld** to the output file.

Absolute External

An absolute symbol that is declared with the `.globl` directive is an absolute external. Its value and type are available to **ld** so that the object file may be linked with others that reference this symbol.

Text (Relocatable)

The value of a .text symbol is measured with respect to the beginning of the .text section of the program. When the assembler output is link-edited, its .text symbols may change in value, because the program may not be the first in the **ld** output. Most .text symbols are defined by appearing as labels. At the start of an assembly, the value of the location counter, specified by a period (.), is "`.text 0`".

Text External

A .text symbol that is declared with the `.globl` directive is a .text external. Its value and type are available to **ld** so that the object file may be linked with others that reference this symbol.

Data (Relocatable)

The value of a .data symbol is measured with respect to the origin of the .data section of a program. The value may change during a subsequent **ld** run because previously loaded programs may have .data sections.

Data External

A .data symbol that is declared with the `.globl` directive is a .data external. Its value and type are available to **ld** so that the object file may be linked with others that reference this symbol.

Shared Data (Relocatable)

The value of a .shdata symbol is measured with respect to the origin of the .shdata section of a program. The value may change during a subsequent **ld** run because previously loaded programs may have .shdata sections.

Shared Data External

A .shdata symbol that is declared with the `.globl` directive is a .shdata external. Its value and type are available to **ld** so that the object file may be linked with others that reference this symbol.

bss (Relocatable)

The value of a .bss symbol is measured from the beginning of the .bss section of a program. The value may change during a subsequent **ld** run, because previously loaded programs may have .bss sections.

bss External

A .bss symbol that is declared with the `.globl` directive is a .bss external. Its value and type are available to **ld** so that the object file may be linked with others that reference this symbol.

Shared bss (Relocatable)

The value of a .shbss symbol is measured with respect to the origin of the .shbss section of a program. The value may change during a subsequent **ld** run because previously loaded programs may have .shbss sections.

Shared bss External

A .shbss symbol that is declared with the `.globl` directive but is defined in the current assembly is a .shbss external. Its value and type are available to **ld** so that the program may be linked with others that reference this symbol.

Constants

Three types of constants can be used as operands in a statement: integer, floating-point, and string. These are described in the following sections.

Integer. All integer constants are 32 bits wide. Such constants are interpreted as two's-complement numbers. Numbers with less precision than 32 bits are treated as 32-bit quantities.

An octal constant consists of a zero followed by a sequence of digits within the range 0-7.

A decimal constant consists of a sequence of digits within the range 0-9. It is not preceded by a zero.

A hexadecimal constant consists of the characters "0x" (or "0X") followed by a sequence containint any of the digits "0123456789abcdefABCDEF".

A binary constant consists of the characters "0b" (or "0B") followed by a sequence of the digits 0 or 1.

Floating-Point. A floating-point constant represented by a decimal value can begin with an optional prefix that specifies its type: "`0f`" or "`0F`" for 4-byte real numbers, "`0d`" or "`0D`" for 8-byte reals. The prefix is followed by an optional sign, then a string of digits optionally containing a decimal point, then an optional "e" or "E" followed by an optionally signed integer:

$$\left[\begin{matrix}\texttt{0f}\\ \texttt{0F}\\ \texttt{0d}\\ \texttt{0D}\end{matrix}\right]\left[\begin{matrix}+\\ -\end{matrix}\right] integer \left[.\left[fraction\right]\right]\left[\left\{\begin{matrix}\texttt{e}\\ \texttt{E}\end{matrix}\right\}\left[\begin{matrix}+\\ -\end{matrix}\right] exponent\right]$$

NOTE

Note that the sign, if any, must follow the prefix, as in this example: `0f-1.3`. ***The assembler uses the library routine*** `atof()` ***to convert floating-point constants.***

The following values are valid representations of floating-point values expressed in decimal:

```
-3.7
-2.5e-7
0f23e98275
```

Floating-point values can also be expressed in hexadecimal. A 4-byte real is expressed as the prefix "`0fx`" (float) followed by 8 hex digits. An 8-byte real is expressed as the prefix "`0dx`" (double) followed by 16 hex digits. The "`f`", "`d`", and "`x`" in the prefix can be lowercase or uppercase. The prefix is not optional when floating-point values are represented in hexadecimal.

The following values are valid representations of floating-point values expressed in hexadecimal:

```
0fx-3.7
0dx23eab27c22943494
```

Floating-point values can also be stored with the `.double` directive. Refer to Chapter 2 for an example of this usage.

Floating-point constants are internally represented in IEEE-standard floating-point format.

String. A string constant is defined using the same syntax and semantics as the C language. Strings begin and end with a double quote ("). All C backslash conventions are observed. Strings are known by their value and their length; the assembler implicitly ends strings with a null byte. A string can contain an arbitrary number of characters.

1.5.2 Operators

Table 1-1 contains a list of operators and their meanings.

Table 1-1
Expression Operators

Operator	Meaning
+	addition
-	(binary) subtraction
*	multiplication
\/	division
&	bitwise logical AND
~	bitwise logical NOT
|	bitwise logical OR
>	right shift
<	left shift
\%	remainder operator
!	bitwise logical AND-NOT

An example of the ! operator is: `a ! b`. A bitwise NOT operation is performed on `b`; this result is used in a bitwise AND operation with `a`. The instruction `-1 ! b` is equivalent to a bitwise NOT operation on `b`.

1.5.3 Evaluating Expressions

The assembler evaluates operators from left to right. Square brackets ([ ]) establish precedence. The assembler evaluates the portion of an expression enclosed in brackets before evaluating the surrounding parts of the expression.

1.5.4 Combining Types in Expressions

The type of the result of an expression depends on two factors: the operator and the types of the operands.

The rules for combining operands in expressions are as follows:

- Two absolute operands can be combined using any of the operators; the type of the result is absolute.

- An absolute operand can be combined with one of the other types using the + or – operators. The type of the result is the same as the type of the other operand.
- Two operands of the same type may be combined with the – operator; the type of the result is absolute.
- Any other combinations of operand types and operators are illegal.

Chapter 2
Assembler Directives

Introduction 2-1
- Abort Assembler Execution (.ABORT) 2-3
- Align Current Location Counter (.align) 2-3
- Generate Packed 80-Bit Decimal Value (.bcd) 2-3
- Assemble into Bss Memory Section (.bss) 2-3
- Store 8-bit Value (.byte) 2-4
- Reserve Global Storage (.comm) 2-4
- Assemble into Data Memory Section (.data) 2-5
- Provide Symbol Definition (.def/.endef) 2-5
  - Specify Array Dimensions (.dim) 2-6
  - Provide Source Line Number (.line) 2-6
  - Provide Storage Class of Expression (.scl) 2-7
  - Provide Size of Expression (.size) 2-7
  - Define Tag Field of a Structure Definition Symbol (.tag) 2-7
  - Define Type of a Symbol (.type) 2-7
  - Define Value of a Symbol (.val) 2-7
- Store 64-bit Floating-point Value (.double) 2-8
- Align Current Location Counter (.even) 2-8
- Specify Source File Name (.file) 2-8
- Store 32-bit Floating-point Value (.float) 2-8
- Define Global (External) Symbol (.globl) 2-9
- Store Source File Identification String in .comment Section (.ident). 2-9
- Store Begin Jump Table Flag Value (.jmpbeg) 2-9
- Store End Jump Table Flag Value (.jmpend) 2-9
- Reserve Local Storage in .bss Memory Section (.lcomm) 2-10
- Provide Line Number Relative to Beginning of Function (.ln) 2-10
- Store 32-bit Integer Value (.long) 2-10
- Turn Off Branch Optimization (.noopt) 2-10
- Turn On Branch Optimization (.optim) 2-11
- Display Path (.path) 2-11
- Assemble into User-Specified Memory Section (.section) 2-11
- Set Symbol to a Value (.set) 2-13
- Store String (.string) 2-13
- Assemble into Text Memory Section (.text) 2-13
- Store 16-Bit Integer Value (.value) 2-14

Store C-Compiler Version String in .comment Section (`.version`).. 2-14

Tables

2-1 Assembler Directives 2-1

Chapter 2
Assembler Directives

Introduction

This chapter contains a table that shows the assembler directives and gives a brief description of each directive's function. Following the table, the function and syntax of each directive is explained in more detail.

Table 2-1
Assembler Directives

Function	Directive	Description
abort assembly	`.ABORT`	abort assembler execution
debugger information	`.file`	specify source file name for error messages and for debugger use
	`.path`	specify source file pathname for debugger use
	`.ln`	specify source line number relative to beginning of function
	`.line`	specify source line number of beginning of function
	`.def/.endef`	specify symbol definition
	`.scl`	specify storage class of symbol
	`.size`	specify size of symbol
	`.tag`	specify tag field of a structure definition symbol
	`.type`	specify type of a symbol
	`.val`	specify value of a symbol
	`.dim`	specify array dimensions

Table 2-1
Assembler Directives (cont.)

Function	Directive	Description
memory sections	`.bss`	assemble into .bss memory section
	`.data`	assemble into .data memory section
	`.text`	assemble into .text memory section
	`.section`	assemble into user-specified memory section
symbol definition	`.globl`	specify global (external) symbol
	`.set`	set symbol to a value
reserve storage	`.comm`	specify global (external) symbol and allocate .bss storage
	`.lcomm`	specify local symbol and allocate .bss storage
alignment	`.align`	align current location counter
	`.even`	align current location counter to an even memory location
branch optimization	`.noopt`	turn off branch optimization
	`.optim`	turn on branch optimization
store data	`.bcd`	generate packed 80-bit decimal value
	`.byte`	store 8-bit value
	`.double`	store 64-bit floating-point value
	`.float`	store 32-bit floating-point value
	`.jmpbeg`	mark begin jump table with flag value
	`.jmpend`	mark end jump table with flag value
	`.long`	store 32-bit integer value
	`.value`	store 16-bit integer value
	`.string`	store string
comments	`.ident`	store identification string in .comment section
	`.version`	store compiler version string in .comment section

The remainder of this section contains detailed descriptions of the assembler directives, including syntax and usage. For ease of reference, descriptions are arranged alphabetically with the exception of the directives that must appear between `.def` and `.endef`. These directives appear in a subgrouping following the `.def` directive.

2.1.1 Abort Assembler Execution (.ABORT)

The .ABORT directive terminates assembly immediately, without creating any files. The correct syntax is:

```
.ABORT
```

In a pipe-interconnected version of a compiler, you can have the compiler issue the .ABORT directive at the first major syntax error, thus saving the assembler from assembling code that is not usable.

2.1.2 Align Current Location Counter (.align)

The .align directive adjusts the location counter upward so that its value is divisible by *n*. The correct syntax is:

```
.align n
```

The argument *n* must be defined and absolute. Its value must be either 2 or 4. The assembler places zero-filled bytes or nop instructions in the current memory section until the location counter value is evenly divisible by *n*.

2.1.3 Generate Packed 80-Bit Decimal Value (.bcd)

The .bcd directive generates a packed 80-bit decimal value into the current .text, .data, or user-specified (but not .bss) section. The correct syntax is:

```
.bcd n
```

The value *n* is a non-floating-point constant.

2.1.4 Assemble into Bss Memory Section (.bss)

The .bss directive switches assembly to the .bss memory section. The correct syntax is:

```
.bss
```

The .bss name directive allocates .bss storage without changing the current section. The correct syntax is:

```
.bss name [, expr]
```

The symbol *name* is any valid symbol that names the allocated memory. The value of the expression *expr* must be a positive integer.

2.1.5 Store 8-bit Value (.byte)

The .byte directive stores an 8-bit value in memory (but not in .bss). The correct syntax is:

```
.byte expr [, expr]...
```

The expression *expr* is an absolute, 8-bit value to be stored. When you specify more than one value, values are stored in successive locations.

2.1.6 Reserve Global Storage (.comm)

The .comm directive reserves global storage. The correct syntax is:

```
.comm name, n
```

The symbol *name* is defined as a global symbol; *n* bytes of global storage are reserved for *name*.

Implementation Details. The symbol *name* is an undefined external until it is defined elsewhere by being used as a .text or .data symbol in the current assembly or by any of the other object files input to the link editor **ld**. However, if *name* is never defined as a .text or .data symbol, *n* bytes of .bss section memory are allocated and *name* is assigned the location of the first byte.

```
.comm name, n, -3
```

The symbol *name* is defined as a global symbol; *n* bytes of global storage are reserved for *name*. The symbol *-3* is a flag to the assembler to use the .shdata and .shbss sections.

Implementation Details. The symbol *name* is an undefined external until it is defined elsewhere by being used as a .text or .shdata symbol in the current assembly or by any of the other object files input to the link editor **ld**. However, if *name* is never defined as a .text or .shdata symbol, *n* bytes of .shbss section memory are allocated and *name* is assigned the location of the first byte.

The following version of the `.comm` directive is used for the .shdata symbol definition and .shbss memory allocation.

2.1.7 Assemble into Data Memory Section (`.data`)

The `.data` directive assembles into the indicated .data subsection. The correct syntax is one of the following:

```
.data
```

```
.data [digit]
```

The optional *digit* can be either 0, 1, 2, or 3. You can specify one of these values to select one of four subsections in the .data section. When you do not specify a .data subsection, the default is .data 0.

Subsections are part of the .data subsection; they are not separate areas of memory. At the end of the assembly, the assembler concatenates all subsections into one .data section. Subsections are useful for creating tables.

Note that if the **–r** assembly flag is specified, the .data section is concatenated to the .text section at the end of the assembly process.

2.1.8 Provide Symbol Definition (`.def/.endef`)

The `.def/.endef` directive along with other debug directives are used to specify sysmbol table debug information. The correct syntax is the following:

```
.def name

    various debug directives

.endef
```

Several debug directives are used to define COFF symbol table entries. These directives must appear only between the `.def`/`.endef` directives. The following list names these directives:

- `.dim`
- `.line`
- `.scl`
- `.size`
- `.tag`
- `.type`
- `.val`

The next subsections describe these directives. For information not provided here, refer to the Common Object File Format section in the operating system programming guide.

Specify Array Dimensions (`.dim`)

The `.dim` directive specifies from 1 to 4 array dimensions. This directive defines the `x_dimen` field of the auxiliary symbol table. This field is declared as an array of 4 unsigned shorts. The correct syntax is:

```
.dim n [, n]
```

The number *n* is a positive integer. Values of $n > 2^{16}$ are rounded down to $2^{16} - 1$.

Provide Source Line Number (`.line`)

The `.line` directive provides a source line number that is used in debugging. This directive defines the `x_lnno` field of the auxiliary symbol table. This field is declared as an unsigned short. The correct syntax is:

```
.line n
```

The number *n* is a positive integer.

Provide Storage Class of Expression (`.scl`)

The `.scl` directive provides the storage class of a specified expression. This directive defines the `n_sclass` field of the symbol table. This field is declared as a char. The correct syntax is:

```
.scl expr
```

Provide Size of Expression (`.size`)

The `.size` directive defines the `x_size` field of the auxiliary symbol table. This field is declared as an unsigned short. The correct syntax is:

```
.size expr
```

Define Tag Field of a Structure Definition Symbol (`.tag`)

The `.tag` directive defines the `x_tgndx` field of the auxiliary symbol table. This field is declared as a long int. The correct syntax is:

```
.tag name
```

Define Type of a Symbol (`.type`)

The `.type` directive defines the `n_type` field of the symbol table. This field is declared as an unsigned short. The correct syntax is:

```
.type expr
```

Define Value of a Symbol (`.val`)

The `.val` directive defines the `n_value` field of the symbol table. This field is declared as a long int. The correct syntax is:

```
.val expr
```

2.1.9 Store 64-bit Floating-point Value (`.double`)

The `.double` directive stores a 64-bit, floating-point constant value in memory (but not in .bss). The correct syntax is:

```
.double n [, n]...
```

When you specify more than one value, values are stored in successive locations.

2.1.10 Align Current Location Counter (`.even`)

The `.even` directive aligns the current location counter to an even memory location. The correct syntax is:

```
.even
```

2.1.11 Specify Source File Name (`.file`)

The `.file` directive provides the source file name used in debugging. The correct syntax is:

```
.file "name"
```

For correct debugging, source files should begin with the `.file` directive. The source file *name* must be a string constant. You can make *name* up to 1024 characters long.

2.1.12 Store 32-bit Floating-point Value (`.float`)

The `.float` directive stores a 32-bit, floating-point, constant value in memory (but not in .bss). The correct syntax is:

```
.float n [, n]...
```

When you specify more than one value, values are stored in successive locations.

2.1.13 Define Global (External) Symbol (.globl)

The .globl directive defines a global symbol; that is, a symbol that is external to the current program unit. The correct syntax is:

```
.globl name
```

If the symbol *name* is otherwise defined (with the .set directive or as a label) in the current module, the assembler treats the symbol as local, but allows other linked modules to refer to it. If the symbol is undefined in the current module, the assembler assumes that it is global.

2.1.14 Store Source File Identification String in .comment Section (.ident)

The .ident directive stores a source file identification string in the .comment section. The correct syntax is:

```
.ident "string"
```

2.1.15 Store Begin Jump Table Flag Value (.jmpbeg)

The .jmpbeg directive stores the begin jump table flag value (0xf1). The correct syntax is:

```
.jmpbeg
```

2.1.16 Store End Jump Table Flag Value (.jmpend)

The .jmpend directive stores the end jump table flag value (0xffff). The correct syntax is:

```
.jmpend
```

2.1.17 Reserve Local Storage in .bss Memory Section (.lcomm)

The .lcomm directive reserves local storage in the .bss portion of the .data section. The correct syntax is:

```
.lcomm name [, n]
```

The local symbol *name* is assigned the location of the first byte. The positive integer *n* determines how many bytes are allocated.

2.1.18 Provide Line Number Relative to Beginning of Function (.ln)

The .ln directive provides the line number, relative to the beginning of a function, that is used in debugging. The correct syntax is:

```
.ln line [, addr]
```

2.1.19 Store 32-bit Integer Value (.long)

The .long directive stores a 32-bit integer in memory (but not in

```
.long n [, n]...
```

The argument *n* is an absolute value. When you specify more than one value, values are stored in successive locations.

2.1.20 Turn Off Branch Optimization (.noopt)

The .noopt directive turns off branch optimization. Branch optimization is the selection of the short form of branch instructions whenever possible. By default branch optimization occurs. The correct syntax is:

```
.noopt
```

Branch optimization can also be turned off by assembling your source file with the **-n** option.

2.1.21 Turn On Branch Optimization (.optim)

The .optim directive turns branch optimization back on after a .noopt directive. The correct syntax is:

```
.optim
```

2.1.22 Display Path (.path)

The .path directive is used to inform the debugger of the path to the original source file if this file is not located in the current directory. The correct syntax is:

```
.path "pathname"
```

Example.

The following command:

```
cc ../common/func.c
```

would result in

```
.path "../common/func.c"
```

2.1.23 Assemble into User-Specified Memory Section (.section)

The .section directive makes it possible for you to define up to five memory sections in addition to the predefined .text, .data, .bss, and .comment sections. When the assembler encounters the *.section* directive, it creates a user-specified section if one does not already exist and makes it the current section. The correct syntax is one of the following:

```
.section name, "attr_string"

.section name
```

The symbol *name* is your name for the section. The *attr_string* is a string containing one or more of the letters from the following list. It defines the

type of section you are creating. You must specify *attr_string* the first time you reference a user-specified section. Thereafter, *attr_string* can be omitted; however, if you specify it in subsequent references, be sure that *attr_string* is the same in all references.

The following list shows the *attr_string* options and their section type. For more information on section types, refer to the Common Object File Format section of the operating system programming guide.

b	STYP_BSS: zero-initialized .data (.bss) section
c	STYP_COPY: copy section (not allocated, not relocated, loaded)
d	STYP_DSECT: dummy section
i	STYP_INFO: information (.comment) section (not allocated, relocated, or located)
l	STYP_LIB: library section (not allocated, relocated, or loaded)
n	STYP_NOLOAD: no-load section
o	STYP_OVER: overlay section
s	STYP_SHARED: shared memory section
w	STYP_DATA: .data (writable) section
x	STYP_TEXT: executable .text section

Example. This example shows a user-specified shared data section called .shdata.

```
.section .shdata, "sw"
```

2.1.24 Set Symbol to a Value (.set)

The .set directive enters a symbol into the symbol table and assigns it a name and a value. The correct syntax is:

```
.set name, expr
```

2.1.25 Store String (.string)

The .string directive stores a string at the current location terminating the string with a null byte. The correct syntax is:

```
.string "string"
```

To include a double quote character in a string, preface each double quote with a backslash character.

2.1.26 Assemble into Text Memory Section (.text)

The .text directive assembles into the indicated .text subsection. The correct syntax is:

```
.text [digit]
```

The optional *digit* can be either 0, 1, 2, or 3. You can specify one of these values to select one of four subsections in the .text section. The default is 0.

Subsections are part of the .text subsection; they are not separate areas of memory. At the end of assembly, the assembler concatenates all subsections into one .text section.

Note that if the **-r** assembly flag is specified, the .data section is concatenated to the .text section at the end of the assembly process.

2.1.27 Store 16-Bit Integer Value (`.value`)

The `.value` directive stores a 16-bit integer value in memory (but not in .bss). The correct syntax is:

```
.value expr [, expr]...
```

The expression *expr* is an absolute, 16-bit value to be stored. When you specify more than one value, values are stored in successive locations.

2.1.28 Store C-Compiler Version String in .comment Section (`.version`)

The `.version` directive stores the C-compiler version string in the

```
.version "string"
```

Chapter 3
80386 Instruction Summary

3.1 Introduction 3-1

3.2 Notational Conventions 3-1

3.3 Operands 3-2
Addressing Modes 3-4
Sample Expressions and Addressing Modes 3-5

3.4 Condition Codes 3-6

3.5 Classified List of Instructions 3-8
String Instructions 3-8
Bit Instructions 3-8
Data Movement Instructions 3-9
Data Movement with Zero or Sign-Extension Instructions 3-9
I/O Instructions 3-9
Flag Instructions 3-9
Bit Rotation Instructions 3-10
Jump and Loop Instructions 3-10
Arithmetic/Logical Instructions 3-11
Conversion Instructions 3-11
Decimal Arithmetic Instructions 3-12
Procedure Call and Return Instructions 3-12
Interrupt Instructions 3-12
Protection Model Instructions 3-12
Segment Register Instructions 3-13
Coprocessor Instructions 3-13
Miscellaneous Instructions 3-13

3.6 Differing Mnemonics 3-14

3.7 80386 Instructions 3-15

Figures

3-1 Sample valid expressions and addressing modes 3-5
3-2 Sample invalid expressions 3-6

Tables

3-1 80386 Registers 3-3
3-2 Addressing Modes 3-4
3-3 Condition Codes 3-6
3-4 Differing Mnemonics 3-14

Chapter 3
80386 Instruction Summary

3.1 Introduction

This chapter summarizes the 80386 instruction set, including addressing modes and restrictions on instruction operands. For more information about the instruction set, refer to the Intel *80386 Programmer's Reference Manual.* Note, however, that the Symmetry assembler differs from the Intel assembler in the following ways:

- The Symmetry assembler prefaces a register name with a percent sign (%) to distinguish register names from symbols.
- The Symmetry assembler uses the syntax: *instruction source, destination*, while Intel uses the syntax: *instruction destination, source*.
- The Symmetry assembler derives its type information from the op-code suffix (b, w, or l), while the Intel 80386 assembler can derive it from operand types.
- The Symmetry assembler uses different mnemonics for some instructions. See Section 3.6 for a list of different mnemonics.

Although the Symmetry assembler supports segment overrides, the DYNIX/ptx operating system does not support the use of segment registers.

3.2 Notational Conventions

The following notational conventions are used to describe mnemonics for the 80386 instruction set:

{ } Select a suffix option. Suffix options are separated from one another by vertical bars (|).

imm Use an immediate value.

reg Use a general-purpose register (listed in Table 3-1).

mem Use a byte, word, or long word memory operand.

r/m Use a byte, word, or long word memory operand or general-purpose register.

creg Use a control register (listed in Table 3-1).

dreg Use a debug register (listed in Table 3-1).

sreg Use a segment register (listed in Table 3-1).

treg Use a test register (listed in Table 3-1).

disp Give the number of bits for a relative jump. The assembler supports only a 32-bit address space, so only 8-bit sign-extended and 32-bit addresses are supported. Generally, *disp* is a text label.

cc Use a condition code, required for `j`*cc* (jump) and `set`*cc* instructions (condition codes are listed in Section 3.4).

instr Use an instruction.

3.3 Operands

Three kinds of operands are generally available to an instruction: memory, immediate, and register operands. Memory is accessed as bytes, words, or long words. Although operands need not occur on their natural boundaries, your source code may incur time penalties if they do not.

An immediate value is either a symbol or a constant preceded by a dollar sign ($). If an immediate value is a constant, its data type is absolute; if it is a symbol, its data type is relocatable. An immediate value cannot be the destination operand of an instruction.

Byte, word, and long word registers are available on the processor. The code segment, instruction pointer, and flag register are not available as explicit operands to instructions. The names of 80386 registers are given in Table 3-1.

Indirect operands, specified with an asterisk (*), are available only to jump and call instructions.

The assembler assumes it is generating code for a 32-bit segment, so it automatically generates the 16-bit data prefix byte when 16-bit data is called for (for example, movw %ax, %bx). For more information on mixing 16-bit code and 32-bit code, refer to the Intel 80386 reference manual.

Table 3-1
80386 Registers

Type	Long	Word	Byte	Byte
General	%eax %edx %ecx %ebx %esp (stack ptr) %ebp (frame ptr) %esi (source index) %edi (destination index)	%ax %dx %cx %bx %sp %bp %si %di	%ah %dh %ch %bh	%al %dl %cl %bl
Segment		%cs (code) %ss (stack) %ds (data) %es (data) %fs (data) %gs (data)		
Control	%cr0 %cr2 %cr3			
Debug	%db0 %db1 %db2 %db3 %db6 %db7			
Test	%tr6 %tr7			

3.3.1 Addressing Modes

Table 3-2 shows the syntax of the addressing modes.

Table 3-2
Addressing Modes

Addressing Mode	Syntax
Register	*reg* (see Table 3-1)
Immediate	a constant, or an expression evaluating to a constant or relocatable value; preceded by dollar sign ($)
Register Relative	*offset*(*reg*) (see Table 3-1) *offset*(%ebp) *offset*(%esp)
Scaled Indexed	[*offset*]([*base*][,*index*][,*scale*])
Indirect	**reg* **reg_relative* **scaled_index*

The address, scale, and offsets required by an addressing mode are specified with constants or relocatables. The base and index are registers.

The immediate addressing mode cannot be used as the base addressing mode for scaled index addressing.

3.3.2 Sample Expressions and Addressing Modes

Figure 3-1 shows examples of valid expressions and addressing modes.

```
.data
.lcomm TABLE1,100
.lcomm TABLE2,200

.text
movl TABLE1,%eax              / load the four-byte value at
                              / address TABLE1 into %eax

movl $TABLE1,%eax             / load the address TABLE1
                              / into %eax

movl $0,%eax                  / load zero into %eax

movl [TABLE1+4],%eax          / load the second four-byte
                              / value in TABLE1 into %eax

movl $[TABLE2-TABLE1],%eax    / load the length of TABLE1
                              / into %eax

jmp  *%eax                    / jump to the address in %eax

jmp  *(%eax)                  / jump indirectly through %eax
                              / (jump to the address that is
                              / stored at the address in
                              / %eax

movl 8(%ebp),%eax             / load %eax from the location
                              / whose address is 8 more than
                              / the address in %ebp

movl (%ebp),%eax              / load %eax from the location
                              / whose address is in %ebp
```

Figure 3-1. Sample valid expressions and addressing modes.

Figure 3-2 shows examples of invalid expressions.

```
movl $TABLE2-$TABLE1,%eax   / taking the difference
                            / between two immediate values
                            / is illegal

movl (TABLE1+0x20),%eax     / using parentheses instead of
                            / square brackets is a
                            / syntactic error
```

Figure 3-2. Sample invalid expressions.

3.4 Condition Codes

The `jcc` and `setcc` instructions require a condition code as a suffix. Table 3-3 lists the codes, their meanings, and the conditions tested. The status flags are cf (carry flag), pf (parity flag), af (adjust flag), zf (zero flag), sf (sign flag), and of (overflow flag).

Table 3-3
Condition Codes

Condition Code	Description	Condition Tested
a nbe	above not below or equal to	(cf or zf) = 0
ae nb nc	above or equal not below not carry	cf = 0
b c nae	below carry not above or equal to	cf = 1

Table 3-3
Condition Codes (cont.)

Condition Code	Description	Condition Tested
be na	below or equal not above	(cf or zf) = 1
e z	equal zero	zf = 1
ne nz	not equal not zero	zf = 0
g nle	greater not less than or equal to	((sf xor of) or zf) = 1
ge nl	greater than or equal to not less than	(sf xor of) = 0
l nge	less than not greater than or equal to	(sf xor of) =1
le ng	less than or equal to not greater than	((sf xor of) or zf) = 1
o	overflow	of = 1
no	not overflow	of = 0
np po	not parity parity odd	pf = 0
p pe	parity parity even	pf = 1
s	sign	sf = 1
ns	not sign	sf = 0
cxz ecxz	test %cx test %ecx	cx = 0 cx = 0

3.5 Classified List of Instructions

This section lists the 80386 instruction set by functional groups. Refer to Section 3.7, "80386 Instructions," for descriptions of the instructions.

3.5.1 String Instructions

cmps
cmpsd
cmpsl
lods
movs
rep
repnz
repz
scas{b | w | l}
scasd
scmp{b | w | l}
slod{b | w | l}
smov{b | w | l}
ssca{b | w | l}
ssto{b | w | l}
stos{b | w | d}
xlat

3.5.2 Bit Instructions

bsf{w | l}
bsr{w | l}
bt{w | l}
btc{w | l}
btr{w | l}
bts{w | l}
shld{w | l}
shrd{w | l}

3.5.3 Data Movement Instructions

leal
mov{b | w | l}
pop{w | l}
popa{w | l}
push{b | w | l}
pusha{w | l}
xchg{b | w | l}

3.5.4 Data Movement with Zero or Sign-Extension Instructions

movsb{w | l}
movswl
movzb{w | l}
movzwl

3.5.5 I/O Instructions

in{b | w | l}
ins{b | w | l}
out{b | w | l}
outs{b | w | l}

3.5.6 Flag Instructions

clc
cld
cli
clts
cmc
lahf
popf{w | l}
popfd
pushf{w | l}
sahf
stc
sti
std

3.5.7 Bit Rotation Instructions

rcl{b|w|l}
rcr{b|w|l}
rol{b|w|l}
ror{b|w|l}

3.5.8 Jump and Loop Instructions

jcc
  ja
  jae
  jb
  jbe
  jc
  jcxz
  je
  jecxz
  jg
  jge
  jl
  jle
  jna
  jnae
  jnb
  jnbe
  jnc
  jne
  jng
  jnge
  jnl
  jnle
  jno
  jnp
  jns
  jnz
  jo
  jp
  jpe
  jpo
  js
  jz
jmp

loop
loope
loopne
loopnz
loopz

3.5.9 Arithmetic/Logical Instructions

adc{b | w | l}
add{b | w | l}
and{b | w | l}
bound{w | l}
cmp{b | w | l}
dec{b | w | l}
div{b | w | l}
idiv{b | w | l}
imul{b | w | l}
inc{b | w | l}
mul{b | w | l}
neg{b | w | l}
not{b | w | l}
or{b | w | l}
sal{b | w | l}
sar{b | w | l}
sbb{b | w | l}
shl{b | w | l}
shr{b | w | l}
sub{b | w | l}
test{b | w | l}
xor{b | w | l}

3.5.10 Conversion Instructions

cbtw
cltd
cwtd
cwtl

3.5.11 Decimal Arithmetic Instructions

aaa
aad
aam
aas
daa
das

3.5.12 Procedure Call and Return Instructions

call
enter
leave
ret

3.5.13 Interrupt Instructions

int
int 3
into
iret
iretd

3.5.14 Protection Model Instructions

arpl
lar{w | l}
lgdt
lidt
lldt
lmsw
lsl{w | l}
ltr
sgdt
sidt
sldt
smsw
str
verr
verw

3.5.15 Segment Register Instructions

lds{w | l}
les{w | l}
lfs{w | l}
lgs{w | l}
lsl{w | l}
lss{w | l}

3.5.16 Coprocessor Instructions

wait

3.5.17 Miscellaneous Instructions

addr16
data16
hlt
lock
nop
set*cc*

3.6 Differing Mnemonics

Table 3-4 lists the Symmetry mnemonics that differ from their Intel 80386 counterparts.

Table 3-4
Differing Mnemonics

Symmetry Mnemonic	Intel Mnemonic
cbtw	cbw
cltd	cdq
cwtd	cwd
cwtl	cwde
movsb	movs
movswl	movsx
movzb or movzwl	movzx
repnz	repne or repnz
repz	repe or repz
scmp or cmps	cmps
slod or lods	lods
smov or movs	movs
ssca or scas	scas
ssto or stos	stos
xlat	xlatb

3.7 80386 Instructions

aaa		ASCII adjust %al after addition
aad		ASCII adjust %ax before division
aam		ASCII adjust %ax after multiplication
aas		ASCII adjust %al after subtraction
adc{b|w|l}	*imm,r/m*	add with carry *imm* to *r/m*
	r/m,reg	add with carry *r/m* to *reg*
	reg,r/m	add with carry *reg* to *r/m*
add{b|w|l}	*reg,r/m*	add *reg* to *r/m*
	imm,r/m	add *imm* to *r/m*
	r/m,reg	add *r/m* to *reg*
addr16		force word address mode
and{b|w|l}	*imm,r/m*	AND *imm* to *r/m*
	r/m,reg	AND *r/m* to *reg*
	reg,r/m	AND *reg* to *r/m*
arpl	*reg,r/m*	adjust RPL of *r/m* to not less than RPL of *reg*
bound{w|l}	*mem,reg*	check array index in *reg* against bounds in *mem*
bsf{w|l}	*r,reg/m*	bit scan forward

bsr{w|l}	*r,reg/m*	bit scan reverse
bt{w|l}	*imm,r/m*	save bit in carry flag
	reg,r/m	save bit in carry flag
btc{w|l}	*imm,r/m*	save bit in carry flag and complement
	reg,r/m	save bit in carry flag and complement
btr{w|l}	*imm,r/m*	save bit in carry flag and reset
	reg,r/m	save bit in carry flag and reset
bts{w|l}	*imm,r/m*	save bit in carry flag and set
	reg,r/m	save bit in carry flag and set
call	**r/m*	call indirect
	disp	call direct
cbtw		convert signed byte in %al to signed word in %ax; the Intel instruction is cbw
clc		clear carry flag
cld		clear direction flag
cli		clear interrupt flag
cltd		convert signed long in %eax to signed 64-bit integer in %edx:%eax; the Intel instruction is cdq
clts		clear task-switched flag

cmc		complement carry flag
cmp{b|w|l}	*imm,r/m*	set flags by subtracting *imm* from *r/m*, leaving operands unchanged
	r/m,reg	set flags by subtracting *r/m* from *reg*, leaving operands unchanged
	reg,r/m	set flags by subtracting *reg* from *r/m*, leaving operands unchanged
cmps		same as scmpl
cmpsd		same as cmps
cmpsl		same as cmps
cwtd		convert signed word in %ax to signed long in %dx:%ax; the Intel instruction is cwd
cwtl		convert signed word in %ax to long in %eax; the Intel instruction is cwde
daa		decimal adjust %al after addition
das		decimal adjust %al after subtraction
data16		force word data mode
dec{b|w|l}	*r/m*	decrement *r/m* by 1
div{b|w|l}	*r/m*	unsigned divide register by *r/m*; %al = quotient and %ah = remainder, or %(e)ax = quotient and %(e)dx = remainder

enter	*imm,imm*	make a stack frame; the first operand specifies the stack frame size and the second gives the lexical nesting order (the operands are in the same order as Intel operands)
hlt		halt
idiv{b|w|l}	*r/m*	signed divide register by *r/m*; %al = quotient and %ah = remainder, or %(e)ax = quotient and %(e)dx = remainder
imul{b|w|l}	*r/m*	%ax = %al * *r/m* byte or %dx:%ax = %ax * *r/m*
	imm,r/m,reg	*reg* = *r/m* * *imm*
	imm,reg	*reg* = *reg* * *imm*; this is the same as *imm, reg, reg*
in{b|w|l}	%dx	input from %dx into register
	imm	input from *imm* into register
inc{b|w|l}	*r/m*	increment *r/m* by 1
ins{b|w|l}		input byte, word, or long from port in %dx to string %edi
int	*imm*	call to interrupt procedure
int 3		interrupt 3
into		interrupt 4
iret		interrupt return

iretd		interrupt return
jcc	*disp*	jump if condition is met; see Section 2.4 for condition codes
jcxz	*disp*	jump if %ecx = 0; same as jecxz
jecxz	*disp*	jump if %ecx = 0; same as jcxz
jmp	*disp*	jump
	**r/m*	jump indirect
lahf		load flags into %ah
lar{w|l}	*r/m,reg*	load access rights byte into register
lds{w|l}	*mem,reg*	load %lds and *reg* with pointer from memory
leal	*mem,reg*	store *mem*'s effective address in *reg*
leave		high level procedure exit
les{w|l}	*mem,reg*	load %es and *reg* with pointer from memory
lfs{w|l}	*mem,reg*	load %fs and *reg* with pointer from memory
lgdt	*mem*	load Global Descriptor Table Register
lgs{w|l}	*mem,reg*	load %gs and *reg* with pointer from memory
lidt	*mem*	load Interrupt Descriptor Table Register

lldt	*r/m*	load Local Descriptor Table Register
lmsw	*r/m*	load machine status word
lock		assert LOCK # signal for next instruction (the set of instructions is limited)
lods		same as slodl
loop	*disp*	decrement count; jump if count <> 0
loope	*disp*	decrement count; jump if count <> 0 and zero flag = 1
loopne	*disp*	decrement count; jump if count <> 0 and zero flag = 0
loopnz	*disp*	decrement count; jump if count <> 0 and zero flag = 0
loopz	*disp*	decrement count; jump if count <> 0 and zero flag = 1
lsl{w|l}	*r/m,reg*	load segment limit into register
lsl		same as lsll
lss{w|l}	*mem,reg*	load %ss and *reg* with pointer from memory
ltr	*r/m*	load task register
mov{b|w|l}	*imm,r/m*	move *imm* to *r/m*
	r/m,reg	move *r/m* to *reg*
	reg,r/m	move *reg* to *r/m*

movl	*creg,reg*	move control register to *reg*
	dreg,reg	move debug register to *reg*
	treg,reg	move test register to *reg*
	reg,creg	move *reg* to control register
	reg,dreg	move *reg* to debug register
	reg,treg	move *reg* to test register
movs		same as smovl
movsb{w|l}	*r/m,reg*	move byte with sign-extend; the Intel instruction is movsx
movswl	*r/m,reg*	move word to long with sign-extend; the Intel instruction is movsx
movw	*r/m,sreg*	move *r/m* to segment register
	sreg,r/m	move segment register to *r/m*
movzb{w|l}	*r/m ,reg*	move byte with zero-extend; the Intel instruction is movzx
movzwl	*r/m,reg*	move byte to long with zero-extend; the Intel instruction is movzx
mul{b|w|l}	*r/m*	unsigned multiply; %ax = %al * *r/m* or %(e)dx:%(e)ax = %(e)ax * *r/m*
neg{b|w|l}	*r/m*	two's complement negate of *r/m*
nop		no operation

not{b|w|l}	*r/m*	reverse each bit of *r/m*
or{b|w|l}	*imm,r/m*	OR *imm* to *r/m*
	r/m,reg	OR *r/m* to *reg*
	reg,r/m	OR *reg* to *r/m*
out{b|w|l}	%dx	output from register to port in %dx
	imm	output from register into *imm*
outs{b|w|l}		output byte, word, or long to port in %dx from string %edi
pop	*sreg*	pop top of stack into segment register
pop{w|l}	*r/m*	pop a word from the stack into *r/m*
popa{w|l}		pop all word/long general registers
popf{w|l}		pop word on stack top into the flags register
popfd		pop long on stack top into the eflags register
push	*sreg*	push segment register
push{b|w|l}	*imm*	push *imm*; push sign extends the immediate byte to a long and pushes a long onto the stack
push{w|l}	*r/m*	push *r/m*

pusha{w|l}		push word/long general registers onto the stack
pushf{w|l}		push the flags register onto the stack
rcl{b|w|l}	%cl,*r/m*	rotate bits and carry flag left, %cl times
	imm,r/m	rotate bits and carry flag left, *imm* times
rcr{b|w|l}	%cl,*r/m*	rotate bits and carry flag right, %cl times
	imm,r/m	rotate bits and carry flag right, *imm* times
rep *instr*		repeat the following string instruction the number of times in the count register
repnz *instr*		repeat the following string instruction while not equal; Intel also calls this repne
repz *instr*		repeat the following string instruction while equal; Intel also calls this repe
ret		return to caller
	imm	return to caller; pop *imm* bytes off the stack
rol{b|w|l}	%cl,*r/m*	rotate bits left, %cl times
	imm,r/m	rotate bits left, *imm* times
ror{b|w|l}	%cl,*r/m*	rotate bits right, %cl times
	imm,r/m	rotate bits right, *imm* times
sahf		store %ah into flags

sal{b|w|l}	%cl,*r/m*	multiply *r/m* by 2, %cl times
	imm,r/m	multiply *r/m* by 2, *imm* times
sar{b|w|l}	%cl,*r/m*	signed divide *r/m* by 2, %cl times
	imm,r/m	signed divide *r/m* by 2, *imm* times
sbb{b|w|l}	*imm,r/m*	subtract with borrow *imm* from *r/m*
	r/m,reg	subtract with borrow *r/m* from *reg*
	reg,r/m	subtract with borrow *reg* from *r/m*
scas{b|w|l}		same as ssca{b|w|l}
scasd		same as sscal
scmp{b|w|l}		compare the byte, word, or long pointed to by %esi to the location pointed to by %edi; the Intel instruction is cmps
set*cc*	*r/m*	set byte on condition (see Section 2.4)
sgdt	*mem*	store Global Descriptor Table Register
shl{b|w|l}	%cl,*r/m*	multiply *r/m* by 2, %cl times
	imm,r/m	multiply *r/m* by 2, *imm* times
shld{w|l}	*imm,reg,r/m*	*r/m* gets SHL of *r/m* concatenated with *reg*; *imm* contains the shift count
	reg,r/m	*r/m* gets SHL of *r/m* concatenated with *reg*; %cl contains the shift count

shr{b|w|l}	%cl,*r/m*	unsigned divide *r/m* by 2, %cl times
	imm,r/m	unsigned divide *r/m* by 2, *imm* times
shrd{w|l}	*imm,reg,r/m*	*r/m* gets SHR of *r/m* concatenated with *reg*; *imm* contains the shift count
	reg,r/m	*r/m* gets SHR of *r/m* concatenated with *reg*; %cl contains the shift count
sidt	*mem*	store Interrupt Descriptor Table Register
sldt	*r/m*	store Local Descriptor Table Register
slod{b|w|l}		load %al, %ax, or %eax with the contents of the location pointed to by %esi; the Intel instruction is lods
smov{b|w|l}		move the byte, word, or long pointed to by %esi to the location pointed to by %edi; the Intel instruction is movs
smsw	*r/m*	store machine status word
ssca{b|w|l}		compare %al, %ax, or %eax with the contents at the location pointed to by %edi; the Intel instruction is scas
ssto{b|w|l}		store %al, %ax, or %eax in the location pointed to by %edi; the Intel instruction is stos
stc		set carry flag to 1
std		set direction flag to 1
sti		set interrupt flag to 1

stos{b | w | d}		same as ssto{b | w | l}
str	*r/m*	store task register
sub{b | w | l}	*imm,r/m*	subtract *imm* from *r/m*
	r/m,reg	subtract *r/m* from *reg*
	reg,r/m	subtract *reg* from *r/m*
test{b | w | l}	*imm,r/m*	AND *imm* with *r/m* and store condition codes only
	r/m,reg	AND *r/m* with *reg* and store condition codes only
	reg,r/m	AND *reg* with *r/m* and store condition codes only
verr	*r/m*	verify segment for reading
verw	*r/m*	verify segment for writing
wait		wait until BUSY pin is inactive (high)
xchg{b | w | l}	*r/m,reg*	exchange *reg* with *r/m*
	reg,r/m	exchange *reg* with *r/m*
xlat		set %al to the memory byte addressed by the sum of the contents of %ebx and %al; the Intel instruction is xlatb

xor{b|w|l}	*imm,r/m*	exclusive OR *imm* to *r/m*
	r/m,reg	exclusive OR *r/m* to *reg*
	reg,*r/m*	exclusive OR *reg* to *r/m*

Chapter 4
80387 FPU Instruction Summary

4.1 Introduction 4-1

4.2 Notational Conventions 4-1

4.3 Operands 4-2

4.4 Classified List of Instructions 4-3
Data Transfer Instructions 4-3
Addition Instructions 4-4
Subtraction Instructions 4-4
Multiplication Instructions 4-4
Division Instructions 4-4
Other Arithmetic Instructions 4-4
Comparison Instructions 4-5
Transcendental Instructions 4-5
Constant Instructions 4-5
Processor Control Instructions 4-6

4.5 80387 FPU Instructions 4-7

Tables

4-1 Instruction Formats 4-3

Chapter 4
80387 FPU Instruction Summary

4.1 Introduction

This chapter summarizes the 80387 floating-point unit instructions. For more information about the instructions, see the Intel *80387 Programmer's Reference Manual.*

NOTE

When %st is the source operand, the following 80387 instructions behave differently than they do with other source operands: `fsub`, `fsubp`, `fsubr`, `fsubrp`, `fdiv`, `fdivp`, `fdivr`, and `fdivrp`. To counteract this problem, the Symmetry assembler provides the following instructions, which behave the same for all source operands: `ffsub`, `ffsubp`, `ffsubr`, `ffsubrp`, `ffdiv`, `ffdivp`, `ffdivr`, and `ffdivrp`. A description of each instruction is provided in Section 4.5.

4.2 Notational Conventions

The following conventions are used to describe mnemonics for the 80387 instruction set:

{}	Select a suffix option. Suffix options are separated from one another by vertical bars (|).
[]	Select a suffix or operand option if necessary. Suffix or operand options are separated from one another by vertical bars (|).
l|s|t	For instructions on real numbers, select *l* for long (64 bits), *s* for short (32 bits), or *t* for temporary real (80 bits).

l l l l For instructions on integers, select *l* for short (32 bits) or *ll* for long (64 bits).

4.3 Operands

A typical instruction operates on one or two operands. An operand is usually the contents of a register or a memory location, and it may be predefined. The operand for the `fsqrt` instruction, for example, always comes from the top element of the floating-point stack, which is %st(0). Some operands may (or must) be coded explicitly with the instruction, and others are implicit. In some cases, both explicit and implicit operands are used, and the implicit operand is usually %st(0). For instructions that have a memory operand, %st(0) is one operand or the only operand.

The registers are named %st(0) through %st(7). The stack top register, %st(0), can also be specified as %st.

Many instructions allow their operands to be coded in more than one way. For example, `fadd` (add real) may be written without operands, with a source, or with a source and a destination. In this section, optional operands are shown in brackets, with alternative operand forms shown with vertical bars (these bars are not coded). For example, the operands for `fadd` are shown in this way:

[*src* | *src,dest*]

This means that `fadd` can be written in any of the three ways shown in Table 4-1.

Table 4-1
Instruction Formats

Written Form	Action
fadd	Add %st(0) to %st(1), then pop %st(0)
fadd *src*	Add source to %st(0)
fadd *src,dest*	Add source to destination

The assembler can allow the same instruction to be specified in different ways. The following example shows two ways to specify the `fadd` instruction:

```
fadd
```

or

```
fadd %st,%st(1)
```

4.4 Classified List of Instructions

This section arranges the 80387 floating-point unit instructions by functional groups. Refer to Section 4.5 for descriptions of the instructions.

4.4.1 Data Transfer Instructions

fbld
fbstp
fild
fist[l]
fistp[l|ll]
fld[l|s|t]
fnop
fst[l|s]
fstp[l|s|t]
fxch

4.4.2 Addition Instructions

fadd[l | s]
faddp
fiadd[l]

4.4.3 Subtraction Instructions

fisub[l]
fsub[l | s] or ffsub[l | s]
fisubr[l]
fsubp or ffsubp
fsubr[l | s] or ffsubr[l | s]
fsubrp or ffsubrp

4.4.4 Multiplication Instructions

fimul[l]
fmul[l | s]
fmulp

4.4.5 Division Instructions

fdiv[l | s] or ffdiv[l | s]
fdivp or ffdivp
fdivr[l | s] or ffdivr[l | s]
fdivrp or ffdivrp
fidiv[l]
fidivr[l]

4.4.6 Other Arithmetic Instructions

fabs
fchs
fscale
fprem
fprem1
frndint
fsqrt
fxtract

4.4.7 Comparison Instructions

fcom[l | s]
fcomp[l | s]
fcompp
ficom[l]
ficomp[l]
ftst
fucom
fucomp
fucompp
fxam

4.4.8 Transcendental Instructions

f2xm1
fcos
fpatan
fptan
fsin
fsincos
fyl2x
fyl2xp1

4.4.9 Constant Instructions

fld1
fldl2e
fldl2t
fldlg2
fldln2
fldpi
fldz

4.4.10 Processor Control Instructions

fclex
fdecstp
ffree
fincstp
finit
fldcw
fldenv
fnop
frstor
fsave
fstcw
fstenv
fstsw
fstsw %ax
fwait

4.5 80387 FPU Instructions

This section contains an alphabetical list of of 80387 floating-point unit instructions with brief descriptions.

f2xm1		compute the function Y = 2 sup X – 1 and replace %st with the result; X = %st and Y = %st(1)
fabs		change %st to its absolute value by making its sign positive
fadd[l|s]	[*src*|*src,dest*]	add two real numbers; add the source and destination operands and return the sum to the destination; %st must be one of the operands
faddp	[*src,dest*]	add two real numbers and pop the stack; %st must be one of the operands
fbld	*src*	convert the content of the source operand from packed decimal to extended real and push the result onto the stack, preserving the sign of the source (including negative zero)
fbstp	*dest*	convert the contents of %st to a packed decimal integer, store the result in the destination in memory, and pop the stack
fchs		complement (reverse) the sign of %st
fclex		clear all exception flags, the exception status flag, and the busy flag in the status word
fcom[l|s]	[*src*]	compare real; set flags based on %st – *src* or %st – %st(1)
fcomp[l|s]	[*src*]	compare real and pop the stack

fcompp		compare real and pop twice, discarding both operands; %st is always compared with %st(1)
fcos		replace the contents of %st with cos(%st)
fdecstp		subtract 1 from the stack pointer in the status word
fdiv[l|s]	[*src*|*src,dest*]	divide real; divide the destination by the source and return the quotient to the destination, unless %st(0) is the source. In this case, divide the source by the destination and return the quotient to the destination. If there are two operands, %st(0) must be one of them.
ffdiv[l|s]	[*src*|*src,dest*]	divide real; divide the destination by the source and return the quotient to the destination. If there are two operands, %st(0) must be one of them.
fdivp	[*src,dest*]	divide real and pop; divide the destination by the source and pop the stack, unless %st(0) is the source. In this case, divide the source by the destination and pop the stack. If there are two operands, %st(0) must be one of them.
ffdivp	[*src,dest*]	divide real and pop; divide the destination by the source and pop the stack. If there are two operands, %st(0) must be one of them.
fdivr[l|s]	[*src*|*src,dest*]	divide real reversed; divide the source by the destination and return the quotient to the destination, unless %st(0) is the source. In this case, divide the destination by the source and return the quotient to the destination. If there are two operands, %st(0) must be one of them.

ffdivr[l | s]	[*src* | *src,dest*]	divide real reversed; divide the source by the destination and return the quotient to the destination. If there are two operands, %st(0) must be one of them.
fdivrp	[*src,dest*]	divide real reversed and pop; divide the source by the destination and pop the stack, unless %st(0) is the source. In this case, divide the destination by the source and pop the stack. If there are two operands, %st(0) must be one of them.
ffdivrp	[*src,dest*]	divide real reversed and pop; divide the source by the destination and pop the stack. If there are two operands, %st(0) must be one of them.
ffree	*dest*	change the destination register's tag to empty (without changing the content of the register)
fiadd[l]	*src*	add two integers
ficom[l]	*src*	convert the source operand (which may refer to a word or short binary integer variable) to extended real and compare %st to it
ficomp[l]	*src*	integer compare and pop, discarding the value in %st by popping the stack
fidiv[l]	*src*	integer divide
fidivr[l]	*src*	integer divide reversed
fild[l | ll]	*src*	convert the source memory operand from its binary-integer format (word, short, or long) to extended-real format and push the result onto the stack

fimul[l]	*src*	integer multiply
fincstp		add 1 to the stack top pointer in the status word
finit		initialize the 80387 processor
fist[l]	*dest*	round %st to an integer according to the round mode specified by the control word and transfer the result to the destination
fistp[l | ll]	*dest*	round %st to an integer (like fist) and pop the stack following the transfer
fisub[l]	*src*	integer subtract
fisubr[l]	*src*	integer subtract reversed
fld[l | s | t]	*src*	push the source operand onto %st
fld1		push +1.0 onto the stack
fldcw	*src*	replace the current processor control word with the word defined by the source
fldenv	*src*	reload the environment from the memory area defined by the source
fldl2e		push $LOG sub 2 e$ onto the stack
fldl2t		push $LOG sub 2 10$ onto the stack
fldlg2		push $LOG sub 10 2$ onto the stack
fldln2		push $LOG sub e 2$ onto the stack

fldpi		push PI onto the stack
fldz		push +0.0 onto the stack
fmul[l|s]	[*src*|*src,dest*]	multiply two real numbers; multiply the source and destination operands and return the product to the destination; %st must be one of the operands
fmulp	[*src,dest*]	multiply real and pop; %st must be one of the operands
fnop		no operation
fpatan		partial arctangent; compute the function arctangent (%st(1) + %st), then pop %st once and replace the new stack top with the result
fprem		compute the remainder of division of %st by %st(1) and leave the result in %st
fprem1		compute the remainder of division of %st by %st(1) and leave the result in %st
fptan		partial tangent; compute the function Y = tan(%st); Y replaces %st, then the value 1 is pushed, becoming the new stack top
frndint		round %st to an integer according to the round mode specified by the control word
frstor	*src*	reload the complete state of the 80387 from the memory area defined by the source
fsave	*dest*	write the full state of the 80387 (environment plus register stack) to the memory location defined by the destination

fscale		convert the value in %st(1) into an integer and add this value to the exponent of the number in %st; this provides rapid multiplication or division by integral powers of 2
fsin		replace the contents of %st with sin(%st)
fsincos		replace the contents of %st with sin(%st), then push cos(%st) onto the stack
fsqrt		replace the contents of %st with its square root
fst[l | s]	*dest*	copy %st to the destination, which may be another register on the stack or a single or double memory location
fstcw	*dest*	write the processor control word to the destination in memory
fstenv	*dest*	write the 80387's basic status (control, status, tag words, and exception pointers) to the destination in memory
fstp[l | s | t]	*dest*	copy %st to the destination, but pop the stack after the transfer
fstsw	*dest*	store the contents of the status word in the destination
fstsw %ax		write the current value of the 80387 status word directly into the 80386 %ax register

fsub[l|s]	[*src*|*src,dest*]	subtract real; subtract the source from the destination and return the difference to the destination, unless %st(0) is the source. In this case, subtract the destination from the source. If there are two operands, %st(0) must be one of them.
ffsub[l|s]	[*src*|*src,dest*]	subtract real; subtract the source from the destination and return the difference to the destination. If there are two operands, %st(0) must be one of them.
fsubp	[*src,dest*]	subtract real and pop; subtract the source from the destination and pop the stack, unless %st(0) is the source. In this case, subtract the destination from the source. If there are two operands, %st(0) must be one of them.
ffsubp	[*src,dest*]	subtract real and pop; subtract the source from the destination and pop the stack. If there are two operands, %st(0) must be one of them.
fsubr[l|s]	[*src*|*src,dest*]	subtract real reversed; subtract the destination from the source and return the difference to the destination, unless %st(0) is the source. In this case, subtract the source from the destination and return the difference to the destination. If there are two operands, %st(0) must be one of them.
ffsubr[l|s]	[*src*|*src,dest*]	subtract real reversed; subtract the destination from the source and return the difference to the destination. If there are two operands, %st(0) must be one of them.

fsubrp	[*src,dest*]	subtract real reversed and pop; subtract the destination from the source and pop the stack, unless %st(0) is the source. In this case, subtract the source from the destination and pop the stack. If there are two operands, %st(0) must be one of them.
ffsubrp	[*src,dest*]	subtract real reversed and pop; subtract the destination from the source and pop the stack. If there are two operands, %st(0) must be one of them.
ftst		test %st by comparing it to zero
fucom	[*src*]	unordered compare real
fucomp	[*src*]	unordered compare real and pop
fucompp		unordered compare real and pop twice, discarding both operands; %st is always compared with %st(1)
fwait		an alternate mnemonic for the 80386 wait instruction
fxam		report the contents of %st
fxch	[*dest*]	swap the contents of the destination and %st
fxtract		decompose the number in %st into two numbers that represent the actual value of the operand's exponent and significand (mantissa) fields; the exponent replaces the original operand on the stack and the significand is pushed onto the stack

fyl2x	compute the function Z = Y * log sub 2 X; X is taken from %st and Y is taken from %st(1)
fyl2xp1	compute the function Z = Y * log sub 2 (X + 1); X is taken from %st and Y is taken from %st(1); X is popped and Z replaces Y at the new stack top

Chapter 5
Programming Example

5.1 Introduction 5-1
5.2 Factorial Numbers 5-1

Figures

5-1 A factorial routine with a C interface 5-2

Chapter 5
Programming Example

5.1 Introduction

This chapter contains a sample assembly language module, which is designed to be linked with a C program. For a description of the stack frame format used by C programs, refer to the C manual.

5.2 Factorial Numbers

The following sample module illustrates how a *naive* assembly language routine (one created without knowledge of the C calling conventions) can be adapted for a C program. The procedure returns any factorial number that can be represented by a four-byte integer. The factorial of number *n* is the product of

```
1 * 2 * 3 * . . . *  n
```

The naive routine, `as_factorial`, returns the factorial of a given number in %eax. If the number has a factorial that cannot be represented as a long integer, the routine returns the error message –1.

When a C program includes a call such as the following:

```
m = factorial (n);
```

the argument to the function (*n*) is passed on the stack, not in a register. Therefore, this example includes a second function, `factorial`, which copies *n* from the stack to %eax and invokes `as_factorial`. The function `factorial` returns the error message –1 if *n* is outside the range determined by the function `factorial`. Note that factorial does not pop *n* off the stack; that is the duty of the calling program, which typically uses an instruction such as the following:

```
addl $4,%esp.
```

```
    .file  "test.s"

    /"Naive" routine
    /A parameter passed in register %eax is returned as the
    /corresponding factorial number in %eax.

    .data
fac:
    .long 0,1,2,6,24,120,720,5040
    .long 40320,362880,39916800,479001600

    .text
as_factorial:
    cmpl   $12,%eax
    ja error
    movl   fac(%eax,4),%eax
    ret

error:
    movl $-1,%eax
    ret

    /C language interface for factorial routine.  factorial
    /is declared global so that ld knows it can use this
    /factorial to resolve global references.

    .globl factorial
factorial:
    pushl  %ebp
    movl   %esp,%ebp
    movl   8(%ebp),%eax
    call   as_factorial
    leave
    ret
```

Figure 5-1. A factorial routine with a C interface.

Appendix A
Porting Considerations

A.1 Introduction A-1
A.2 Porting a System V i386 Program to DYNIX/ptx A-1
A.3 Porting an i386 Program from DYNIX 3.0 to DYNIX/ptx A-1

Appendix A
Porting Considerations

A.1 Introduction

This appendix addresses problems you may encounter when porting System V i386 assembler programs to a Symmetry system running the DYNIX/ptx operating system or from a Symmetry system running the DYNIX 3.0 (or later) operating system to one running DYNIX/ptx.

A.2 Porting a System V i386 Program to DYNIX/ptx

When porting an i386 assembler program to a Symmetry system running the DYNIX/ptx operating system, make sure that assembler functions that call C or are called from C adhere to the calling protocol documented in the C manual.

In particular, C functions that are compiled with the **-fpa** option look for floating-point return values in the Weitek %fp register, not in the i387 %st(0) register.

A.3 Porting an i386 Program from DYNIX 3.0 to DYNIX/ptx

This section lists the changes you must make when porting an i386 assembler program from a Symmetry system running the DYNIX 3.0 (or later) operating system to a Symmetry system running the DYNIX/ptx operating system. Depending on your application, porting from a system running DYNIX 3.0 (or later) to one running DYNIX/ptx may require changes not listed here.

- In assembler invocations, replace the DYNIX 3.0 **-R** flag with the DYNIX/ptx **-r** flag. The DYNIX 3.0 **-R** flag and the DYNIX/ptx **-r** flag make initialized data read-only. The DYNIX/ptx **-R** flag unlinks the input file.
- Review the invocation descriptions in the DYNIX 3.0 and DYNIX/ptx assembler manuals for any other argument differences.

- Change the comment format, introducing every DYNIX/ptx comment with a forward slash (/), rather than the DYNIX 3.0 comment characters: pound (#), and slash asterisk . . . asterisk slash (/* . . . */).
- Make the first line of the file a `.file` directive. For example:

```
.file "test.s"
```

- Change any `bsf*` and `bsr*` instructions so that source and destination are reversed:

DYNIX 3.0	DYNIX/ptx
`bsfl <dest>,<src>`	`bsfl <src>,<dest>`

When the source is a memory operand, only one form is legal; you will get an assembler error message if the instruction is of the wrong form.

DYNIX 3.0	DYNIX/ptx
`bsfl %eax,_var`	`bsfl _var,%eax`

Be especially careful when both operands are registers. If you do not make the recommended change, you will not get an assembler error message, but the opposite instruction will be generated. For example, the following two instructions have the same meaning:

DYNIX 3.0	DYNIX/ptx
`bsfl %eax,%ecx`	`bsfl %ecx,%eax`

- Change the syntax if you want to retain segment overrides. However, note that segment overrides will have no effect, because DYNIX/ptx does not support segmented memory. For example, the following syntax is acceptable to the assembler:

DYNIX 3.0	DYNIX/ptx
`cs addl lab,%eax`	`addl %cs`

- Change operand size overrides. In DYNIX 3.0, operand size overrides are prefixes, but in DYNIX/ptx they are treated as separate instructions. For example:

DYNIX 3.0	DYNIX/ptx
`data16 movl (%eax),%eax`	`data16; movl (%eax),%eax`

- Change the syntax in the `enter` instruction. For example:

DYNIX 3.0	DYNIX/ptx
`enter $4,4`	`enter $4,$4`

- Change the syntax in the `int` instruction.

DYNIX 3.0	DYNIX/ptx
`int 3`	`int $3`

- Substitute DYNIX/ptx directives for DYNIX 3.0 directives, as follows:

DYNIX 3.0	DYNIX/ptx
`.asciz`	`.string`
`.int`	`.long`
`.line`	`.ln`
`.org n`	`.set .,n`
`.space n`	`.set .,.+n`
`.word`	`.value`
`.align 1`	`.align 2`
`.align 2`	`.align 4`

- Replace the following directives, which are not supported by DYNIX/ptx:

DYNIX 3.0	DYNIX/ptx
`.ascii`	<None>
`.fill`	<None>

- Correct debugger symbol table information as follows (see the DYNIX 3.0 and DYNIX/ptx assembler manuals for more details):

DYNIX 3.0	DYNIX/ptx
`.lsym`	`.def/.endef`
`.stab`	`.def/.endef`--subdirectives are `.type`, `.val`, `.tag`, `size`, `.scl`, `.line`, `.dim`

- Correct shared data and shared bss section directives as follows (see the DYNIX 3.0 and DYNIX/ptx assembler manuals for more details):

DYNIX 3.0	DYNIX/ptx
`.shdata`	`.section .shdata, "ws"`
`.shbss`	`.section .shbss, "bs"`

- Correct instruction mnemonics as follows:

DYNIX 3.0	DYNIX/ptx
`cmpsd`	`cmpsl`
`scasd`	`scasl`
`stosd`	`stosl`
`scasd`	`scas, scasl`
`stosd`	`stos, stosl`

- Precede these operators: asterisk (*), slash (/), and percent (%) with a backslash (). For example:

DYNIX 3.0	DYNIX/ptx
`movl $4*1024,%eax`	`movl $4\*1024,%eax`

- Revise expressions affected by the DYNIX/ptx order of evaluation, as follows:

DYNIX 3.0	DYNIX/ptx
Operators evaluated by precedence. 2+4*3 evaluates to 2+(4*3) or 14	Operators evaluated left to right, except with brackets. 2+4*3 evaluates to (2+4)*3 or 18. 2+[4*3] evaluates to 14

- Replace double-pointed brackets (<< and >>) with single-pointed brackets (< and >). For example:

DYNIX 3.0	DYNIX/ptx
`movl $7 << 4,%eax`	`movl $7 < 4,%eax`

- Replace any uses of tilde (~) with the one's complement of a number. For example:

DYNIX 3.0	DYNIX/ptx
~17	–17 –1

- Replace any uses of exclamation point (!).

DYNIX 3.0	DYNIX/ptx
a!b means a|[~b]	a!b means a&[~b]

- Remove any uses of ^.

DYNIX 3.0	DYNIX/ptx
a^b means exclsv. OR	<No exclusive OR operator> ^ is a typecast operator

Appendix B
1167 Floating-point Accelerator Operation Summary

B.1 Introduction B-1
B.2 Notational Conventions B-1
B.3 Operands B-2
B.4 1167 Floating-point Accelerator Operations B-3

Appendix B
1167 Floating-point Accelerator Operation Summary

B.1 Introduction

This appendix summarizes macros that are patterned after the Weitek 1167 Floating-point Accelerator instruction set. You can expand individual macros into sequences of multiple instructions. The 80386 instruction stream contains 80386 `mov` instructions rather than 1167 instructions. The operands of the 80386 `mov` instructions contain special bit fields that can be interpreted by the 1167. (For information about coding moves explicitly, see documentation for the Weitek WTL 1167 Floating-point Coprocessor Board.)

Macros that use memory operands destroy the contents of the 80386 %eax register.

The 1167 does not operate directly on operands in memory. If a macro operand is a memory location, the macro expansion will include an instruction that loads the operand into the %eax register, where it can be accessed by the 1167. For double-precision memory operands, %eax is typically used twice—once for each half of the operand. To view the instruction sequence produced for a particular macro, use the disassembler, **disas**.

B.2 Notational Conventions

The following notational conventions are used to describe instruction mnemonics:

{s|l}	Select a suffix option, "s" for single precision or "l" for double precision.
r/m	Use a register or memory operand.
freg	Use a floating-point accelerator register.
reg	Use an 80386 general-purpose register.

r/m/f/fimm Use an 80386 general-purpose register, a memory operand, a floating-point accelerator register, or a floating-point immediate operand.

B.3 Operands

The names of the floating-point accelerator registers are %fp2—%fp31. Register %fp1 has special uses not described here. For single-precision operations, any numbered register can be used. For double-precision operations, even numbers represent register pairs. For example, a single-precision operation in %fp3 uses only %fp3. A double-precision operation in %fp4 uses %fp4 and %fp5.

A floating-point immediate operand is represented by $ followed by a floating-point constant. (Refer to Section 1.3.2.)

B.4 1167 Floating-point Accelerator Operations

pushl	*freg*	push *freg* only; to push a register pair, code the operation for each register
popl	*freg*	pop *freg* only; to pop a register pair, code the operation for each register
wabs{s|l}	*r/m/f/fimm,freg*	produce absolute value; *freg* = |*r/m/f/fimm*|
wadd{s|l}	*r/m/f/fimm,freg*	add; *freg* = *freg* + *r/m/f/fimm*
wamul{s|l}	*r/m/f/fimm,freg*	multiply absolute value; *freg* = |*rm/f/fimm*| * |*freg*|
wcmp{s|l}	*r/m/f/fimm,freg*	set 1167 flags based on *r/m/f/fimm* – *freg*; no trap on unordered
wcmpt{s|l}	*r/m/f/fimm,freg*	set 1167 flags based on *r/m/f/fimm* – *freg*; trap on unordered
wcvtls	*r/m/f/fimm,freg*	convert single-precision *r/m/f/fimm* to double-precision and store the result in *freg*
wcvtsl	*r/m/f/fimm,freg*	convert double-precision *r/m/f/fimm* to single-precision and store the result in *freg*
wdiv{s|l}	*r/m/f/fimm,freg*	divide; *freg* = *r/m/f/fimm* + *freg*
wfix{s|l}	*r/m/f/fimm,freg*	convert floating-point to integer; 32-bit-integer result

wfloat{s|l}	*imm/r/m/f,freg*	convert long integer to floating-point; 64-bit-integer result
wldctx	*reg*	load context register; transfer context register to bus
wload{s|l}	*r/m/f/fimm,freg*	load floating-point register
wmacls	*r/m/f/fimm,freg*	multiply and accumulate in registers 2 and 3; single-precision inputs and double-precision result
wmacs	*r/m/f/fimm,freg*	multiply and accumulate in register 2; single-precision inputs and result
wmul{s|l}	*r/m/f/fimm,freg*	multiply; *freg* = *r/m/f/fimm* * *freg*
wmuln{s|l}	*r/m/f/fimm,freg*	multiply and negate; *freg* = – *r/m/f/fimm* * *freg*
wneg{s|l}	*r/m/f/fimm,freg*	negate and add zero; trap on unordered
wrestore	*freg*	pop a series of 1167 registers, beginning with *freg* and ending with %fp31. This destroys the contents of %esi, %edi, and %ecx. The operation is efficient only if you are restoring six or more registers. The macro expands into a sequence of instructions terminating in a string move. This sequence (rather than wrestore) is displayed by the disassembler.

wsave	*freg*	push a series of 1167 registers, beginning with *freg* and ending with %fp31. This destroys the contents of %esi, %edi, and %ecx. The operation is efficient only if you are saving six or more registers. The macro expands into a sequence of instructions terminating in a string move. This sequence (rather than wsave) is displayed by the disassembler.
wstctx	*reg*	copy context register to 80386 register
wstor{s | l}	*freg,r/m*	copy 1167 register to 80386 register or memory
wsub{s | l}	*r/m/f/fimm,freg*	subtract; *freg* = *r/m/f/fimm* – *freg*
wtst{s | l}	*freg*	set 1167 flags based on *freg* – 0; no trap on unordered
wtstt{s | l}	*freg*	set 1167 flags based on *freg* – 0; trap on unordered

Index

.ABORT directive 2-3
.align directive 2-3
.bcd directive 2-3
.bss 2-3
(.bss directive) 2-10
.bss directive 2-3,10
.byte directive 2-4
.comm directive 2-4
.data directive 2-5
.def directive 2-5
.dim directive 2-6
.double directive 2-8
.endef directive 2-5
.even directive 2-8
.file directive 2-8
.float directive 2-8
.globl directive 2-9
.ident directive 2-9
.jmpbeg directive 2-9
.jmpend directive 2-9
.line directive 2-6
.ln directive 2-10
.long directive 2-10
.noopt directive 2-10
.optim directive 2-11
.path directive 2-11
.scl directive 2-7
.section directive 2-11
.set directive 2-13
.size directive 2-7
.string directive 2-13
.tag directive 2-7
.text directive 2-13
.type directive 2-7
.val directive 2-7
.value directive 2-14
.version directive 2-14

1167 Floating-point Accelerator
operations ..... Appendix B
16-bit integer
.value directive 2-14
32-bit integer
.long directive 2-10
64-bit value
.double directive 2-8
8-bit value
.byte directive 2-4
80386
assembler, Intel 3-1
condition codes 3-6
instructions ..... Chapter 3
classified list 3-8
operands 3-2
registers 3-2,3
80387
instructions ..... Chapter 4
classified list 4-4
operands 4-2
registers 4-3

A

aaa 3-12,15
aad 3-12,15
aam 3-12,15
aas 3-12,15
Abort assembly (.ABORT directive) 2-3
Absolute
data 1-9
external 1-9
adc{b|w|l} 3-11,15
Addition instructions 4-5
addr16 3-13,15
add{b|w|l} 3-11,15
Align data (.align directive) 2-3
Alignment
location counter 2-8
and{b|w|l} 3-11,15
Arithmetic instructions 4-5

Arithmetic/logical instructions 3-11
arpl 3-12,15
Arrays
  .dim directive 2-6
as command 1-1
Assembler directives 2-1
Assembler, invocation 1-1

B

Bit instructions 3-8
Bit rotation instructions 3-10
Blanks 1-7
bound{w|l} 3-11,15
branch optimization 1-1
Branch Optimization
  .noopt directive 2-10
  .optim directive 2-11
bsf{w|l} 3-8,15
bsr{w|l} 3-8,16
bss
  external 1-10
  relocatable 1-10,11
Bss section 1-3
btc{w|l} 3-8,16
btr{w|l} 3-8,16
bts{w|l} 3-8,16
bt{w|l} 3-8,16

C

call 3-12,16
cbtw 3-11,16
cbw 3-14
cdq 3-14
clc 3-9,16
cld 3-9,16
cli 3-9,16
cltd 3-11,16
clts 3-9,16
cmc 3-9,17
cmps 3-8,17

cmpsd 3-8,17
cmpsl 3-8,17
cmp{b|w|l} 3-11,17
Comment statement 1-7
Comparison instructions 4-6
Constant instructions 4-6
Constants 1-11
Control register 3-3
Conversion instructions 3-11
Coprocessor instructions 3-13
cwd 3-14
cwde 3-14
cwtd 3-11,17
cwtl 3-11,17

D

daa 3-12,17
das 3-12,17
Data
  absolute 1-9
  align 2-3
  external 1-10
  relocatable 1-10
  section (.data directive) 2-5
  undefined 1-9
  movement instructions 3-9
  movement with zero or sign-extension instructions 3-9
  transfer instructions 4-4
Data section 1-3
data16 3-13,17
Debug register 3-3
dec{b|w|l} 3-11,17
Define symbol (.set directive) 2-13
Directive statement 1-7
Directives 2-1
  .ABORT directive 2-3
  .align directive 2-3
  .bcd directive 2-3
  .bss directive 2-3,10
  .byte directive 2-4
  .comm directive 2-4

.data directive 2-5
.def directive 2-5
.dim directive 2-6
.double directive 2-8
.endef directive 2-5
.even directive 2-8
.file directive 2-8
.float directive 2-8
.globl directive 2-9
.ident directive 2-9
.jmpbeg directive 2-9
.jmpend directive 2-9
.line directive 2-6
.ln directive 2-10
.long directive 2-10
.noopt directive 2-10
.optim directive 2-11
.path directive 2-11
.scl directive 2-7
.section directive 2-11
.set directive 2-13
.size directive 2-7
.string directive 2-13
.tag directive 2-7
.text directive 2-13
.type directive 2-7
.val directive 2-7
.value directive 2-14
.version directive 2-14
Division instructions 4-5
div{b|w|l} 3-11,17
dl flag 1-1

E

enter 3-12,17
Expression(s) 3-5
operators 1-13
type propagation in 1-13
External
bss 1-10
data 1-10

shared bss 1-11
shared data 1-10
undefined 1-9
External symbol (.globl directive) 2-9

F

f2xm1 4-6,8
fabs 4-5,8
Factorial routine 5-1
fadd[l|s] 4-5,8
faddp 4-5,8
fbld 4-4,8
fbstp 4-4,8
fchs 4-5,8
fclex 4-7,8
fcom[l|s] 4-6,8
fcomp[l|s] 4-6,8
fcompp 4-6,9
fcos 4-6,9
fdecstp 4-7,9
fdiv[l|s] 4-5,9
fdivp 4-5,9
fdivr[l|s] 4-5,9
fdivrp 4-5,10
ffdiv[l|s] 4-5,9
ffdivp 4-5,9
ffdivr[l|s] 4-5,10
ffdivrp 4-5,10
ffree 4-7,10
ffsub[l|s] 4-5,14
ffsubp 4-5,14
ffsubr[l|s] 4-5,14
ffsubrp 4-5,15
fiadd[l] 4-5,10
ficom[l] 4-6,10
ficomp[l] 4-6,10
fidiv[l] 4-5,10
fidivr[l] 4-5,10
fild 4-4,10
File formats 1-2
fimul[l] 4-5,11

fincstp 4-7,11
finit 4-7,11
fist[l] 4-4,11
fistp[l|ll] 4-4,11
fisub[l] 4-5,11
fisubr[l] 4-5,11
Flag instructions 3-9
Flags
  jump table 2-9
fld1 4-6,11
fldcw 4-7,11
fldenv 4-7,11
fldl2e 4-6,11
fldl2t 4-6,11
fldlg2 4-6,11
fldln2 4-6,11
fldpi 4-6,12
fldz 4-6,12
fld{l|s|t} 4-4,11
Floating-point number (.float directive) 2-8
fmul[l|s] 4-5,12
fmulp 4-5,12
fnop 4-4,7,12
fpatan 4-6,12
fprem 4-5,12
fprem1 4-5,12
fptan 4-6,12
frndint 4-5,12
frstor 4-7,12
fsave 4-7,12
fscale 4-5,13
fsin 4-6,13
fsincos 4-6,13
fsqrt 4-5,13
fst[l|s] 4-4,13
fstcw 4-7,13
fstenv 4-7,13
fstp[l|s|t] 4-4,13
fstsw 4-7,13
fstsw %ax 4-7,13
fsub[l|s] 4-5,14
fsubp 4-5,14

fsubr[l|s] 4-5,14
fsubrp 4-5,15
ftst 4-6,15
fucom 4-6,15
fucomp 4-6,15
fucompp 4-6,15
Function
  line number 2-10
fwait 4-7,15
fxam 4-6,15
fxch 4-4,15
fxtract 4-5,15
fyl2x 4-6,16
fyl2xpl 4-6,16

G-H

General registers 3-3
Global
  storage (.comm directive) 2-4
hlt 3-13,18

I

I/O instructions 3-9
idiv{b|w|l} 3-11,18
Immediate
  addressing mode 3-4
  operand 3-2
imul{b|w|l} 3-11,18
inc{b|w|l} 3-11,18
Indirect
  addressing mode 3-4
  operand 3-2
Instruction statement 1-8
Instruction(s)
  1167 Floating-point Accelerator B-1
  80386 ..... Chapter 3
    classified list 3-8
  80387 ..... Chapter 4
  addition 4-5
  arithmetic 4-5

arithmetic/logical 3-11
bit 3-8
bit rotation 3-10
comparison 4-6
constant 4-6
conversion 3-11
coprocessor 3-13
data movement 3-9
data movement with zero or sign-extension 3-9
data transfer 4-4
decimal arithmetic 3-12
division 4-5
flag 3-9
I/O 3-9
interrupt 3-12
jump 3-10
miscellaneous 3-13
multiplication 4-5
pointer 3-3
procedure call and return 3-12
processor control 4-7
protection model 3-12
segment register 3-13
string 3-8
subtraction 4-5
transcendental 4-6
`ins{b|w|l}` 3-9,18
`int` 3-12,18
`int 3` 3-12,18
Integer 1-11
`.long` directive 2-10
16-bit 2-14
32-bit 2-10
Interrupt instructions 3-12
`into` 3-12,18
Invoking the assembler 1-1
`in{b|w|l}` 3-9,18
`iret` 3-12,18
`iretd` 3-12,19

J

ja 3-10
jacc 3-10
jae 3-10
jb 3-10
jbe 3-10
jc 3-10
jcc 3-19
jcxz 3-10,19
je 3-10
jecxz 3-10,19
jg 3-10
jge 3-10
jl 3-10
jle 3-10
jmp 3-10,19
jna 3-10
jnae 3-10
jnb 3-10
jnbe 3-10
jnc 3-10
jne 3-10
jng 3-10
jnge 3-10
jnl 3-10
jnle 3-10
jno 3-10
jnp 3-10
jns 3-10
jnz 3-10
jo 3-10
jp 3-10
jpe 3-10
jpo 3-10
js 3-10
Jump instructions 3-10
Jump table
. jmpbeg directive 2-9
. jmpend directive 2-9
flag 2-9
jz 3-10

K-L

k flag 1-1
Labels
  named 1-6
  numeric 1-6
`lahf` 3-9,19
`lar{w|l}` 3-12,19
`lds{w|l}` 3-13,19
`leal` 3-9,19
`leave` 3-12,19
`les{w|l}` 3-13,19
`lfs{w|l}` 3-13,19
`lgdt` 3-12,19
`lgs{w|l}` 3-13,19
`lidt` 3-12,19
Line number
  function 2-10
Line number (`.ln` directive) 2-10
Line numbers
  `.line` directive 2-6
`lldt` 3-12,20
`lmsw` 3-12,20
Location counter 1-4
  `.even` directive 2-8
  alignment 2-8
`lock` 3-13,20
`lods` 3-8,20
`loop` 3-11,20
`loope` 3-11,20
`loopne` 3-11,20
`loopnz` 3-11,20
`loopz` 3-11,20
`lsl` 3-20
`lsl{w|l}` 3-12,13,20
`lss{w|l}` 3-13,20
`ltr` 3-12,20

M

m flag 1-1
Memory operand 3-2

Memory sections 1-3
movl 3-21
movs 3-8,14,21
movsb{w|l} 3-9,21
movswl 3-9,21
movsx 3-14
movw 3-21
movzb{w|l} 3-9,21
movzwl 3-9,21
movzx 3-14
mov{b|w|l} 3-9,20
Multiplication instructions 4-5
mul{b|w|l} 3-11,21

N

n flag 1-1
Named labels 1-6
neg{b|w|l} 3-11,21
nop 3-13,21
not{b|w|l} 3-11,22
Numeric labels 1-6

O

o flag 1-2
Operands
  1167 Floating Point Accelerator B-2
  80386 3-2
  80387 4-2
  immediate 3-2
  indirect 3-2
  memory 3-2
  register 3-2
Operations, 1167 Floating-point Accelerator ..... Appendix B
or{b|w|l} 3-11,22
outs{b|w|l} 3-9,22
out{b|w|l} 3-9,22

P-Q

Packing (.bcd directive) 2-3
Path
  .path directive 2-11
pop 3-22
popa{w|l} 3-9,22
popfd 3-9,22
popf{w|l} 3-9,22
popl B-3
pop{w|l} 3-9,22
Porting A-1
Processor control instructions 4-7
Programming example 5-1
Protection model instructions 3-12
Pseudo-operations 2-1
push 3-22
pusha{w|l} 3-9,23
pushf{w|l} 3-9,23
pushl B-3
push{b|w|l} 3-9,22
push{w|l} 3-22

R

R flag 1-2
r flag 1-2
rcl{b|w|l} 3-10,23
rcr{b|w|l} 3-10
Register(s)
  1167 Floating Point Accelerator B-2
  80386 3-3
  80387 4-3
  addressing mode 3-4
  names 1-5
  operand 3-2
  relative addressing mode 3-4
Relocatable
  bss 1-10
  Data 1-10

relocatable
shared bss 1-11
Relocatable
shared data 1-10
text 1-9
rep 3-8,23
repe 3-14
repne 3-14
repnz 3-8,23
repz 3-8,23
ret 3-12,23
Reversed instructions
division 4-5
subtraction 4-5
rol 3-23
rol{b|w|l} 3-10
ror 3-23
ror{b|w|l} 3-10

S

sahf 3-9,23
sal{b|w|l} 3-11,24
sar{b|w|l} 3-11,24
sbb{b|w|l} 3-11,24
Scaled indexed addressing mode 3-4
scasd 3-8,24
scas{b|w|l} 3-8,24
scmp{b|w|l} 3-8,24
Sections
.bss 1-3; 2-10
.bss (.bss directive) 2-3
.data 1-3
.text 1-3; 2-13
user-specified 1-4; 2-11
Segment(s)
register instructions 3-13
registers 3-3
setcc 3-13,24
sgdt 3-12,24

shared bss
external 1-11
Shared data
external 1-10
relocatable 1-10
shld{w|l} 3-8,24
shl{b|w|l} 3-11,24
shrd{w|l} 3-8,25
shr{b|w|l} 3-11,25
sidt 3-12,25
Size
.size directive 2-7
sldt 3-12,25
slod{b|w|l} 3-8,25
smov{b|w|l} 3-8,25
smsw 3-12,25
Source file
.ident directive 2-9
identification string 2-9
name 2-8
Source file name
.file directive 2-8
ssca{b|w|l} 3-8,25
ssto{b|w|l} 3-8,25
Statement formats 1-4
Statements
comment 1-7
directive 1-7
instruction 1-8
Status registers 3-3
stc 3-9,25
std 3-9,25
sti 3-9,25
Storage class
.scl directive 2-7
Store String (.string directive) 2-13
stos{b|w|d} 3-8,26
str 3-12,26
String
(.string directive) 2-13
String instructions 3-8

Structure definition
tag 2-7
Subtraction instructions 4-5
sub{b|w|l} 3-11,26
Symbol 1-4
(.set directive) 2-13
external 2-9
type 2-7
types 1-9
value 2-7
Symbol definition
.double directive 2-5

T

Tag
.tag directive 2-7
Test register 3-3
test{b|w|l} 3-11,26
Text
external 1-10
relocatable 1-9
Text section 1-3
(.text directive) 2-13
Transcendental instructions 4-6
Type
.type directive 2-7
Type propagation in expressions 1-13

U

u flag 1-2
Undefined
data 1-9
Undefined external 1-9
User-specified section
(.section directive) 2-11

V

V flag 1-2

Value
.val directive 2-7
verr 3-12,26
Version string
.version directive 2-14
verw 3-12,26

W

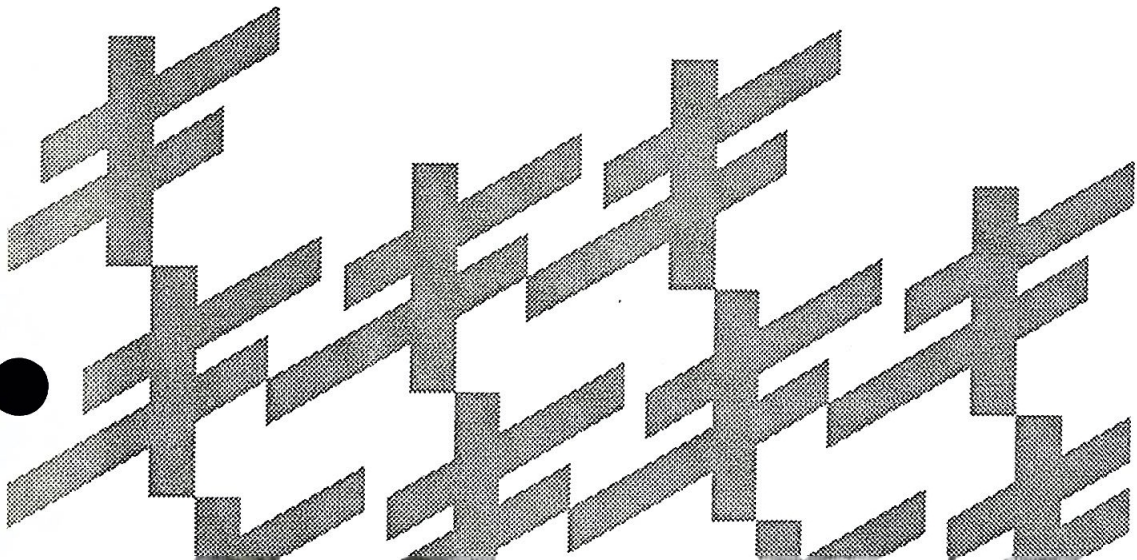
wabs{s/l} B-3
wadd{s|l} B-3
wait 3-13,26
wamul{s|l} B-3
wcmpt{s|l} B-3
wcmp{s|l} B-3
wcvtls B-3
wcvtsl B-3
wdiv{s|l} B-3
wfix{s|l} B-3
wfloat{s|l} B-4
wldctx B-4
wload{s|l} B-4
wmacls B-4
wmacs B-4
wmuln{s|l} B-4
wmul{s|l} B-4
wneg{s|l} B-4
wrestore B-4
wsave B-5
wstctx B-5
wstor{s|l} B-5
wsub{s|l} B-5
wtstt{s|l} B-5
wtst{s|l} B-5

X-Y-Z

xchg{b|w|l} 3-9,26
xlat 3-8,26
xlatb 3-14
xor{b|w|l} 3-11,27
Y flag 1-2

DYNIX/ptx®
Pdbx User's Manual

1003-48547-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

Symmetry and Pdbx are trademarks of Sequent Computer Systems, Inc.

UNIX is a registered trademark of AT&T.

Copyright 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America.

Contents

About This Manual
Organization v
Notational Conventions vi
Related Publications vii

Chapter 1. Introduction
1.1 Overview 1-1
1.2 Overview of Demonstration Programs 1-2
1.3 Demonstration Run: Simple C Program 1-3
1.4 Demonstration Run: Simple Parallel FORTRAN Program 1-13

Chapter 2. Getting Started
2.1 Introduction 2-1
2.2 Before Compiling Your Code 2-1
2.3 Compiling Programs for Use with Pdbx 2-2
2.4 Linking Programs for Use with Pdbx 2-3
2.5 Setting Up Your Debugging Environment 2-3
2.6 Invoking Pdbx 2-5

Chapter 3. Using Pdbx
3.1 Introduction 3-1
3.2 Getting Help 3-1
3.3 Error Recovery 3-2
3.4 Language Considerations While Debugging 3-4
3.5 Referencing Setuid Programs 3-4
3.6 Pdbx Command Line Syntax 3-5
3.7 Common Debugging Tasks 3-6
3.8 Debugging Specialized Applications 3-23

Chapter 4. Debugging Parallel Applications
4.1 Introduction 4-1
4.2 Considerations for Parallel Applications 4-2
4.3 Debugging Heterogeneous Applications 4-6

Chapter 5. Command Dictionary
5.1 Introduction 5-1
5.2 The pdbx Command 5-1
5.3 Pdbx Command Line Syntax 5-3
5.4 Command Descriptions 5-10

Appendix A. Error and Warning Messages

Appendix B. Reserved Words

Appendix C. Differences from UNIX 4.3bsd dbx

C.1 Overview C-1
C.2 pdbx vs. dbx C-3
C.3 Command Line Syntax Changes C-4
C.4 Command Set Modifications C-4
C.5 Using Pdbx as dbx C-7

Figures

1-1 Sample C program with bug 1-4
1-2 Sample FORTRAN program with bug 1-15

C-1 sys.c program listing C-8

Tables

2-1 Pdbx Command Options 2-5

3-1 run vs. create 3-6
3-2 Frequently Used Trace Commands 3-15
3-3 Changing the Radix 3-20

4-1 Comparing C Shell and Pdbx Commands for Terminal I/O 4-4

5-1 Examples of Qualified Identifiers 5-5
5-2 Arithmetic and Boolean Operators 5-7
5-3 Symmetry Register Names 5-9
5-4 Format Letters to Use When Displaying Memory 5-15
5-5 Pdbx-Defined Debugger Variables 5-58

C-1 Command Set Comparison C-4

About This Manual

This manual describes how to use Pdbx, Sequent's parallel debugger, and is intended for use by experienced C, FORTRAN, or 80386/80387 assembly language programmers.

Organization

This manual is organized as follows:

Chapter 1, **Introduction**, provides an overview of Pdbx and presents two sample debugging sessions. The first session uses a simple C program, while the second uses a parallel FORTRAN program.

Chapter 2, **Getting Started**, explains how to run Pdbx and how to initialize your debugging environment.

Chapter 3, **Using Pdbx**, provides step-by-step instructions for common debugging tasks.

Chapter 4, **Debugging Parallel Applications** describes debugging implications for parallel programs.

Chapter 5, **Command Dictionary**, describes each Pdbx command in detail. Commands are listed in alphabetical order for ease of reference. If you don't know the name of a command, refer to the *Commands by Function* list at the beginning of Chapter 5. This list categorizes commands by their function and includes the command name, a brief description of the command, and the page number to reference for detailed information on that command. Chapter 5 also includes information on the debugger's command line syntax.

Appendix A, **Error and Warning Messages**, is an alphabetical listing of common warning and error messages produced by Pdbx. For each warning or error, the message text, an explanation of the message, and a recommended action are listed.

Appendix B, **Reserved Words**, lists the identifiers in Pdbx that have special meaning in some contexts.

Appendix C, **Differences from UNIX 4.3 dbx**, describes the differences between the UNIX 4.3bsd debugger, dbx, and the current version of Pdbx.

If you are familiar with the UNIX 4.3 debugger, you may want to skip ahead to Appendix C for a list of differences between Pdbx and dbx. You may also want to read Chapter 4 for information on debugging parallel programs. Use Chapter 5 or the *Pdbx Quick Reference Card* for command-specific information.

If you are a novice Pdbx user, read Chapters 1–4 carefully and use Chapter 5 and the appendixes for reference. Once you are an experienced Pdbx user, you may also want to use the *Pdbx Quick Reference Card*.

Notational Conventions

This manual uses the following notational conventions:

- The operating system prompt is represented by a percent sign (%).
- The Pdbx prompt is represented by `(pdbx)`.
- Operating system commands and their options, Pdbx commands, and Pdbx command line parameters that are not replaceable values are shown in **bold** type and must be entered exactly as shown. System calls are also shown in **bold** type.
- Program listings and system output are shown in `even-width` font.
- Filenames and values that you must replace are shown in *italics*. *Italics* are also used for emphasis.
- Mutually exclusive items are stacked within brackets ([ ]) or braces ({ }). These items can be a set of commands or a set of parameters. When you *must* choose exactly one command or parameter, they are stacked within braces as in the following:

$$\left\{ \begin{array}{l} sink \\ swim \end{array} \right\}$$

  When commands or parameters are stacked within brackets, you *may* choose one or none, but not both:

$$\left[\begin{array}{l} catsup \\ mustard \end{array} \right]$$

- Brackets enclosing a parameter that is not stacked indicates an optional parameter. When more than one optional parameter exists on a command line, any combination of these parameters can be specified.

- The characters "Ctrl" precede keys that you must press while holding down the control key. For example, Ctrl-C means press the C key while holding down the control key.

Related Publications

The following Sequent documents supplement the contents of this manual:

- Pdbx Quick Reference Card
- Guide to Parallel Programming

Chapter 1
Introduction

1.1 Overview 1-1

1.2 Overview of Demonstration Programs 1-2

1.3 Demonstration Run: Simple C Program 1-3
Overview of C Factorial Program 1-3
Compiling the C Factorial Program 1-5
Debugging the C Factorial Program 1-6

1.4 Demonstration Run: Simple Parallel FORTRAN Program 1-13
Overview of FORTRAN Factorial Program 1-14
Compiling the FORTRAN Factorial Program 1-16
Debugging the FORTRAN Factorial Program 1-17

Figures

1-1 Sample C program with bug 1-4
1-2 Sample FORTRAN program with bug 1-15

Chapter 1
Introduction

1.1 Overview

This chapter provides an overview of Pdbx and presents two sample debugging sessions.

Pdbx is an interactive, symbolic debugger for use in debugging applications written in C, FORTRAN, 80386/80387 assembly language, or any combination of these languages. You can use Pdbx to debug both conventional sequential applications and parallel applications. You can also use Pdbx to debug applications that consist of multiple processes running different programs, such as client/server applications.

The major features of Pdbx are summarized as follows:

- **High-level language debugger.** Pdbx debugs applications written in C, FORTRAN, or 80386/80387 assembly language. Pdbx interprets and displays data types and structures in the language context of the program being debugged. For example, when debugging C programs, you can reference and display pointers, structs, and functions; when debugging FORTRAN programs, you can reference and display arrays, common blocks, and subroutines.

  Using Pdbx, you can trace program execution, monitor changes to individual variables, trace procedure calls and returns, set breakpoints, examine variables, change the values of selected variables, and control and monitor the delivery of signals from the terminal and other processes. Except when stopped at a breakpoint or when tracing, the process being debugged executes just as it would in its normal operating environment,

- **Based on dbx.** Pdbx is based on dbx, the high-level language debugger already in use on many UNIX systems. When used to debug single-process applications, Pdbx performs almost identically to UNIX 4.3bsd dbx. For a list of the differences between Pdbx and UNIX 4.3bsd dbx, refer to Appendix C.

- **Multiple processes.** Pdbx can monitor multiple processes simultaneously. Typically, the processes are related, as in a parent process and multiple child processes in a parallel application, but they may be completely unrelated.

  Pdbx can independently start, stop, trace, or examine processes. You can set breakpoints in individual processes or in all processes. A breakpoint can stop all processes or only the process that encountered the breakpoint.

  Pdbx can run a process in the foreground or background. When a process is running in the foreground, it controls the terminal until it is interrupted, as when you enter Ctrl-C. When a process is running in the background, you can use the terminal to enter Pdbx commands. Pdbx's handling of foreground and background processes is similar to that of the C shell.

  You can direct Pdbx to stop program execution when a process forks (by means of the **fork()** system call), execs (by means of the **execve()** system call), or exits (by means of the **exit()** system call). These events can be "caught" or "ignored" like ordinary UNIX signals.

- **Multiple executable files.** Different processes being debugged can run different programs, or a process can *exec* (that is, stop running the current program and begin running a new program). When a process execs, it assumes the debugging context of the new program, including the current breakpoints, trace options, and source files.

- **Multiple core files.** Multiple core dumps can be examined in the same debugging session. The state of a process that dumped core can be examined in much the same way as a process that is stopped at a breakpoint.

1.2 Overview of Demonstration Programs

To help you get started with Pdbx, the rest of this chapter consists of two sample debugging sessions. The first session demonstrates the debugging of a conventional C program, *fac.c*. The second session demonstrates the debugging of a parallel FORTRAN program, *pfac.f*. Each program contains a bug that you will find using Pdbx. These programs are not intended to be useful, efficient, or robust: they are provided only to illustrate how you can use Pdbx.

Both programs compute the factorial of the number specified on the program's command line. The factorial function is defined as follows:

$$n! = n \times (n-1) \times \cdots \times 1$$

For example, entering **fac 4** should cause the *fac.c* program to display "`4! = 24`", since the following is true:

$$4! = 4 \times 3 \times 2 \times 1 = 24$$

If you specify a number outside the range 1–12, the program prints an error message, since the factorial function is undefined for nonpositive numbers, and 13! is too large to express as a signed 4-byte integer.

When Pdbx exits, it terminates any remaining processes that it created. If you must quit in the middle of either exercise, you may wish to use the operating system **ps** command to check for processes that are no longer associated with a parent process. Use **kill -9** to kill any such processes.

1.3 Demonstration Run: Simple C Program

This debugging session shows how to use Pdbx to debug a simple C program.

NOTE

This example uses the DYNIX/ptx C V1.5p compiler for program development. Using any other compiler or any other version of this compiler may require modifying the program and may result in different displays.

1.3.1 Overview of C Factorial Program

Figure 1-1 lists *fac.c*. This program computes and displays the factorial of the argument passed to the program at run-time. Note that the program has a bug on line 24. For now, ignore the bug.

```
 1  /*
 2   * Usage: fac n
 3   * Computes n factorial (n!) using a recursive
 4   * algorithm. Line 24 contains an intentional error.
 5   */
 6
 7  main (argc, argv)
 8          int     argc;
 9          char    **argv;
10  {
11          int     n;
12
13          n = atoi (argv[1]);
14          if (n<1 || n>12) {
15                  printf ("Number out of range.\n");
16                  exit(1);
17          } else
18                  printf ("%d!=%d\n",n,factorial(n));
19  }
20
21  factorial (n)
22          int     n;
23  {
24          if (n = 1)
25                  return (1);
26          else
27                  return (n * factorial(n-1));
28  }
```

Figure 1-1. Sample C program with bug. Lines 13–16 of **main()** extract *n* from the command line and verify that it is in range. Line 18 prints the values of *n* and *n* factorial. Lines 21–28 are the **factorial()** function, which contains an error on line 24.

In the C program, for values greater than 1, the following is true:

$$n! = n \times (n-1)!$$

Thus, a recursive function calculates the factorial.

1.3.2 Compiling the C Factorial Program

Copy the demonstration program, */usr/sequent/pdbx.demo/fac.c*, into your current directory.

Now compile the program, directing the linked output to *fac*.

```
% cc -g -o fac fac.c Return
```

The **-g** option directs the compiler, assembler, and linker to create the debugging information needed by Pdbx. The **-o** option specifies *fac* as the name of the executable file, the file that contains the executable code for the program.

Run the program by typing the name of the executable file, a space, and the argument.

```
% fac 5 Return
5!=1
```

The answer is incorrect. Run the program several times, using a different argument each time.

```
% fac 2 Return
2!=1
% fac 1 Return
1!=1
```

The results show that the program doesn't work, since the answer is always 1, regardless of the value passed to the program. Let's use Pdbx to debug the program.

1.3.3 Debugging the C Factorial Program

To debug the program whose executable file is *fac*, use the **pdbx** command and specify **fac** as an argument.

```
% pdbx fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx)
```

Pdbx reads in the debugging information from *fac* and creates a dummy process from the *fac* program. Pdbx creates this dummy process, the *program image*, to support features such as code disassembly before you create a process to execute. Since Pdbx assigns each process a unique identifying number, this first process is %1.

To run the program under Pdbx, use the **run** command. The Pdbx **run** command terminates any existing processes (process %1 in this case) and creates a new process (%2) that is run with the specified command-line arguments. For example, entering **run 5** should produce the same result as entering **fac 5** from the operating system.

```
(pdbx) run 5 Return
5!=1
  %2  Stopped by Exit in _exit at 0x314b
0000314b  ret
```

Process %2 runs, yielding the expected erroneous output. By default, Pdbx stops each process just before it terminates (executes the **exit**() system call) and reports the event. To eliminate reporting of termination, enter the following command.

```
(pdbx) ignore exit Return
```

Run the program again to see the effect of the new setting. This time, use the **rerun** command to run the program. Entering the **rerun** command without arguments runs the program using the arguments from the last **run** command.

```
(pdbx) rerun Return
5!=1
```

Note that now there's no difference between what the program displays at the terminal when running under Pdbx and when running from a shell.

Next use the Pdbx **list** command to display the first several lines of **main()**.

```
(pdbx) list main Return
    5    */
    6
    7   main (argc, argv)
    8            int      argc;
    9            char     **argv;
   10   {
   11            int      n;
   12
   13            n = atoi (argv[1]);
   14            if (n<1 || n>12) {
   15                     printf ("Number out of range.\n");
```

Pdbx lists the 11 lines centered around the brace ({) that marks the beginning of the code for **main()**. Note that line 13 extracts the value of *n* from the command line, and that line 14 checks if the value of *n* is less than 1 or greater than 12. Perhaps the program doesn't receive the correct value of *n*. Let's check the value of *n* using the **stop**, **run**, and **print** commands.

```
(pdbx) stop at 14 Return
[1] stop at 14
```

The **stop at 14** command directs Pdbx to set a breakpoint after line 13, but before executing the machine code generated for line 14. Note that Pdbx echoes the breakpoint settings.

Now run the program and print the value of *n* at the breakpoint.

```
(pdbx) run 5 Return
  %4  Stopped at breakpoint 1 in main at line 14
   14             if (n<1 || n>12) {
(pdbx) print n Return
5
```

The program has the correct value of *n*, so the problem must be elsewhere. Use the **cont** command to continue execution of the process after the breakpoint.

```
(pdbx) cont Return
5!=1
```

Once again, the result is incorrect. Since the program is printing the correct value of *n*, and only the line containing the call to the **factorial()** function remains, the problem must reside in **factorial()**. Exercise that procedure by itself:

```
(pdbx) print factorial(5) Return
Process is not stopped
```

When directed to print the value of a function in a program, Pdbx actually executes the code in the function, passing it the specified value(s) on the stack. Since the current process (%4) has exited, Pdbx cannot execute any code. To confirm that the process has exited, use the Pdbx **ps** command. This command reports the status of all processes that have not been terminated. (Remember, entering **run** terminates all previously created processes.)

```
(pdbx) ps Return
* %4  Exited
```

To exercise the **factorial()** function, you need to create a new process. Do this by entering the **create** command.

```
(pdbx) create Return
  %5  Stopped by Exec in . at 0xd0
000000d0  subl   $0x8,%esp
```

Pdbx creates a new process, but stops the process before it executes any code in the program being debugged. Also, unlike **run**, **create** doesn't terminate any other processes.

Now execute **factorial()** and print the result.

```
(pdbx) print factorial(5) Return
1
```

You can create an abbreviation for **print factorial(5)** by using the **alias** command. Pdbx also provides predefined aliases for frequently used commands. To list the currently defined aliases, enter the **alias** command.

```
(pdbx) alias Return
n        next
t        where
j        status
st       stop
[More aliases are printed, but are not listed here]
```

NOTE

The order in which aliases and their definitions are displayed at your terminal may differ from the example shown above. The ordering of aliases for display is program-dependent.

Now create the alias **f(*n*)** to replace the longer string **print factorial(*n*)**.

```
(pdbx) alias f(n) "print factorial(n)" Return
```

Use the alias **f(*n*)** to execute the **factorial** function.

```
(pdbx) f(2) Return
1
(pdbx) f(12) Return
1
```

Running the **factorial()** function independently also yields erroneous results. Let's look at the source for **factorial()**.

```
(pdbx) list factorial Return
   18              printf ("%d!=%d\n",n,factorial(n));
   19    }
   20
   21    factorial (n)
   22            int     n;
   23    {
   24            if (n = 1)
   25                    return (1);
   26            else
   27                    return (n * factorial(n-1));
   28    }
```

Let's set a breakpoint at line 27, where **factorial()** invokes itself to handle values greater than 1.

```
(pdbx) stop at 27 Return
no executable code at line 27
```

Unlike some compilers, the C compiler still performs some optimizations when the debugging option is specified. Since the code at line 27 will never be executed (because of the error at line 24), the C compiler deletes the code. As a result, Pdbx displays the message `no executable code at line 27` when an attempt is made to set a breakpoint at that location.

Since no executable code exists at line 27, the error must occur elsewhere in the program before line 27. Let's trace the **factorial** function using the **trace** command. But first, use the **status** and **delete** commands to display and delete all unnecessary trace settings. The **status** command prints a list of all current stop and trace settings. The **delete** command cancels the specified settings.

```
(pdbx) status Return
[1] stop at 14
(pdbx) delete 1 Return
```

Now use the **trace** command to display the value with which **factorial()** is called, and the value **factorial()** returns.

```
(pdbx) trace factorial Return
[3] trace factorial
(pdbx) run 5 Return
%6  calling factorial(n = 5) from function main
%6  returning 1 from factorial
5! = 1
```

The result shows that values are correctly passed between **main()** and **factorial()**.

Let's use the trace command again to track the value of *n* in **factorial()**.

```
(pdbx) trace n in factorial Return
[4] trace fac.main.n in factorial
```

Wait a minute. Pdbx thinks you want to trace the value of **main()**'s variable *n*, not *n* in **factorial()**.

Precede the variable name with the procedure name to track the variable *n* in **factorial()**.

```
(pdbx) trace factorial.n Return
[5] trace fac.factorial.n in factorial
```

Now cancel the erroneous **trace** command and any other unnecessary settings, and run the program again.

```
(pdbx) status Return
[3] trace factorial
[4] trace fac.main.n in factorial
[5] trace fac.factorial.n in factorial
(pdbx) delete 3 4 Return
(pdbx) run 5 Return
%7 initially (at line 23 in "fac.c"):  fac.factorial.n = 5
%7 after line 24 in "fac.c":   fac.factorial.n = 1
5! = 1
(pdbx)
```

Aha! The value of *n* is changed in line 24. Examine that line using the list command.

```
(pdbx) list 24 Return
   24             if (n = 1)
```

Instead of checking for equality, line 24 assigns the value of 1 to *n*. Replace "`n = 1`" with "`n == 1`" to compare the values of *n* and 1.

Now that you've found the bug, exit Pdbx and (if you like) fix the source file, recompile, and test the fix.

```
(pdbx) quit Return
% vi fac.c Return
  [Replace = with == or <= in line 24.]
% cc -o fac fac.c Return
% fac 5 Return
5!=120
% fac 1 Return
1!=1
% fac 12 Return
12!=479001600
```

1.4 Demonstration Run: Simple Parallel FORTRAN Program

The second debugging session shows how to use Pdbx to debug a parallel program. This example assumes you are familiar with the basic concepts of parallel programming on Sequent systems. For information on this topic, refer to Sequent's *Guide to Parallel Programming*.

NOTE

This example uses Sequent's ptx/FORTRAN V1.8p compiler for program development. Using any other compiler or any other version of this compiler may require modifying the program and may result in different displays.

1.4.1 Overview of FORTRAN Factorial Program

The FORTRAN factorial program uses the parallel programming directive C$DOACROSS to cause parallel execution of the DO loop on lines 24 and 25. Identifying *factorial* as a LOCAL variable results in each child process having a local copy of the variable *factorial*.

Figure 1-2 lists the FORTRAN demonstration program, *pfac.f*. This program's bug is in line 23.

```
1          program pfac
2    c
3          character*8 argbuf
4          integer*4   n, factorial
5    c
6    c  get argument from command line
7    c
8          call getarg (1, argbuf)
9          read (argbuf, 50) n
10   50    format (i8)
11   c
12   c  verify the number is within the allowed range
13   c
14         if (n .lt. 1 .or. n .gt. 12) then
15            write (*,*) 'number out of range.'
16              call fhalt (1)
17         endif
18   c
19   c  calculate the factorial of the number and
20   c  print the result
21   c
22         factorial = 1
23   C$DOACROSS LOCAL(factorial)
24         do 100 i = 1, n
25   100      factorial = factorial * i
26         write (*,*) n, '! = ', factorial
27         end
```

Figure 1-2. Sample FORTRAN program with bug. Lines 8–17 extract *n* from the command line and verify that it is in the range 1–12. Line 22 initializes the value of *factorial* to 1. Lines 23–25 specify a parallel DO loop, which causes child processes to be created during execution. When all iterations have completed, the parent prints the result (line 26) and exits.

1.4.2 Compiling the FORTRAN Factorial Program

Copy the demonstration program, */usr/sequent/pdbx.demo/pfac.f*, into your directory. Compile the program with the **-mp**, **-g**, and **-o** options, and store the linked output in the file *pfac*.

```
% fortran -mp -g -o pfac pfac.f Return
```

The **-mp** option causes the compiler to process parallel directives and restructure code for parallel execution. The **-g** option directs the compiler to create debugging information for Pdbx, and the **-o** option specifies the name of the executable file. Note that the DYNIX/ptx C and ptx/FORTRAN compilers store debugging information in an object file named *filename.o*. The information in *filename.o* is incorporated into the load file (in this case, *pfac*) at link time.

Now run the program.

```
% pfac 1 Return
      1! =           1
% pfac 2 Return
      2! =           1
% pfac 3 Return
      3! =           1
```

The answers are incorrect since 1! = 1, 2! = 2, and 3! = 6. Whenever a parallel program behaves unexpectedly, you should check for and kill orphan processes. These processes are no longer associated with a parent process and use computer resources that may be needed by other processes in the system. Use the DYNIX/ptx **ps** command to list all processes except process group leaders.

```
% ps Return
 PID TTY          TIME COMMAND
 6492 ttyGA/GAAQ  0:01 csh
13979 ttyGA/GAAQ  0:00 ps
```

Your display may have different values in the `PID`, `TTY`, and `TIME` columns. There are no processes running *pfac*, so no orphan processes exist.

1.4.3 Debugging the FORTRAN Factorial Program

Invoke Pdbx to debug *pfac*:

```
% pdbx pfac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
[found symbol  _init_shbrk,assuming parallel library used,ignoring SEGV]
```

The message indicates that a call to **_init_shbrk** was found in the program, causing Pdbx to automatically ignore the SIGSEGV signal. (Parallel programs use SIGSEGV signals to grow stack and data space in shared memory. Refer to Chapter 3 for more information on Pdbx's treatment of SIGSEGV.)

Now direct Pdbx to ignore exits and then run the program. If you do not direct Pdbx to ignore exits, Pdbx reports the event and the session appears to hang. This is because the foreground process is associated with the parent process, and the parent process is waiting for the child (which is stopped) to exit.

```
(pdbx) ignore exit Return
(pdbx) run 3 Return
         3! =          1
```

Running the program under Pdbx yields the same result as entering **pfac 3** from the operating system level. Use the Pdbx **ps** command to check if all processes completed.

```
(pdbx) ps Return
  %3  Exited
* %2  Exited
```

The previous display shows that all processes have exited, and process %2 is the current process. During this execution of the program, processes %2 and %3 executed. Many factors determine the number of child processes created for parallel execution; the only guarantee is that the number of child processes will not exceed the number of processors on the system. (If your system has more processors than DO-loop iterations, your display will differ. The **ps** display will list the message `Running cont` for processes that are not associated with a DO-loop iteration.)

Now list the source program. (Remember, you can start listing from any line in the program by specifying the line number as an argument to the **list** command.)

```
(pdbx) list Return

 1          program pfac
 2    c
 3          character*8 argbuf
 4          integer*4   n, factorial
 5    c
 6    c  get number from command line
 7    c
 8          call getarg (1, argbuf)
 9          read (argbuf, 50) n
10   50    format (i8)
```

List the next 17 lines.

```
(pdbx) list 11,27 (Return)
11  c
12  c  verify the number is within the allowable range
13  c
14        if (n .lt. 1 .or. n .gt. 12) then
15           write(*,*) 'number out of range.'
16           call fhalt(1)
17        endif
18  c
19  c  calculate the factorial of the number and
20  c  print the result
21  c
22        factorial = 1
23  C$DOACROSS LOCAL(factorial)
24        do 100 i = 1,n
25  100      factorial = factorial * i
26        write(*,*) n, '! = ', factorial
27        end
```

Verify that the program is receiving the argument specified on the command line by setting a breakpoint at line 14 and examining the value in *argbuf*.

```
(pdbx) stop at 14 (Return)
[1] stop at 14
(pdbx) run 3 (Return)
  %4  Stopped at breakpoint 1 in pfac.pfac at line 14
   14              if (n .lt. 1 .or. n .gt. 12) then
(pdbx) print argbuf (Return)
'3        '
```

The value in *argbuf* is 3, indicating that the argument is passed correctly.

NOTE

Notice that Pdbx continues to number processes in ascending order. Since the program has run once with Pdbx assigning process numbers of %2 and %3, subsequent processes are %4, %5, and so forth. Use the **ps** *command to see all processes.*

At this point, you could set more breakpoints and run the program again, or use the **next**, **step**, or **cont** command to continue through the program. The **next** and **step** commands execute the next line of code in the current process. The **cont** command continues execution of the current process. For this example, use the **next** command.

```
(pdbx) next Return
  %4  Stopped after next in pfac.pfac at line 22
    22            factorial = 1
```

The value of *factorial* should not yet be 1, because line 22 has not yet executed. Verify the value of *factorial* with the **print** command.

```
(pdbx) print factorial Return
0
```

Continue stepping through the program.

```
(pdbx) next Return
  %4  Stopped after next in pfac.pfac at line 24
   24             do 100  i=1, n
```

The value of *factorial* should be 1, since it was initialized on line 22. Use the **print** command again to display the value of *factorial*.

```
(pdbx) print factorial Return
1
```

Since you cannot step into a parallel DO loop, set a breakpoint at line 25 and continue execution.

NOTE

Do not use the **next**, **step**, *or* **trace** *commands to enter parallel DO loops. When the line to execute contains a call to a procedure for which the source is not available (in this case, a routine in the parallel programming library that starts the child processes running in parallel), these commands execute the procedure to completion.*

```
(pdbx) stop at 25 Return
[2] stop at 25
(pdbx) cont Return
  %5  Stopped at breakpoint 2 in do.100.i.pfac at line 25
   25   100              factorial = factorial * i
  %4  Stopped at breakpoint 2 in do.100.i.pfac at line 25
   25   100              factorial = factorial * i
```

The display shows that the executing processes are %4 and %5. You can check the value of *factorial* in each process. At this point in the program, the value of *factorial* in both processes should be 1.

```
(pdbx) print factorial Return
3
(pdbx) print 5:factorial Return
1908736
```

The value of *factorial* should be 1, because it was initialized before the DO loop and no calculations have yet been performed. If you are familiar with parallel programming as described in Sequent's *Guide to Parallel Programming*, you may have guessed the problem.

Let's execute the program to completion. Use the **status** command to list the current breakpoints and delete the breakpoint at line 25.

```
(pdbx) status Return
[1] stop at 14
[2] stop at 25
(pdbx) delete 2 Return
```

Now enter the **cont** command with the **all** argument to continue execution of all processes. Specifying **cont** only would continue execution of the current process, leaving other processes waiting indefinitely.

```
(pdbx) cont all Return
         3! =              1
(pdbx) quit Return
```

In the C$DOACROSS directive on line 23, *factorial* is declared as a LOCAL variable. As described in the *ptx/FORTRAN User's Manual*, LOCAL variables must be initialized within a loop before they are read. *factorial* is initialized outside of the loop at line 22, so whatever value exists at that location in memory is read. Also, a local variable cannot be used to pass data values in or out of the loop or between loop iterations. The incorrect declaration of *factorial* is the bug.

Now examine statement 100 again. According to the *ptx/FORTRAN User's Manual*, *factorial* is a REDUCTION variable. By declaring *factorial* a REDUCTION variable, the value of the initialized *factorial* is carried into the loop and *factorial* is initialized to 1 for each child process.

Modify the program *pfac.f*, changing LOCAL to REDUCTION in the source code on line 23. Recompile and test the fix.

```
% vi pfac.f Return
[Replace LOCAL with REDUCTION in line 23]
% fortran -mp -o pfac pfac.f Return
    :
    :
% pfac 5 Return
   5! =        120
% pfac 2 Return
   2! =          2
% pfac 1 Return
   1! =          1
% pfac 12 Return
  12! = 479001600
% pfac 3 Return
   3! =          6
```

Chapter 2
Getting Started

2.1 Introduction 2-1

2.2 Before Compiling Your Code 2-1
Source Files Exceeding 100,000 Lines 2-1
vfork Referenced 2-1

2.3 Compiling Programs for Use with Pdbx 2-2
Debugging With Limited Information 2-2

2.4 Linking Programs for Use with Pdbx 2-3

2.5 Setting Up Your Debugging Environment 2-3

2.6 Invoking Pdbx 2-5

Tables

2-1 Pdbx Command Options 2-5

Chapter 2
Getting Started

2.1 Introduction

This chapter describes how to get started using Pdbx. Refer to Chapter 3 for information on using Pdbx *after* it is invoked.

2.2 Before Compiling Your Code

To ensure a successful debugging session, answer the following questions. If you can answer "no" to each question, proceed to Section 2.3. If you answer "yes" to either or both questions, read the appropriate section (Section 2.2.1 or Section 2.2.2) for information on modifying your source code.

- Is your source file greater than 100,000 lines?
- Is **vfork** referenced?

2.2.1 Source Files Exceeding 100,000 Lines

Pdbx does not work with source files exceeding 100,000 lines (this includes comment lines). When the total length of your source file exceeds 100,000 lines and you want to debug your program using Pdbx, you must restructure the program by placing some of the source in a separate file.

2.2.2 vfork Referenced

Pdbx does not support the **vfork**(2) system call because **vfork** does not copy the entire context of the parent. In most programs, replacing calls to **vfork** with calls to **fork** (refer to **fork**(2)) retains program functionality and allows successful debugging. Be sure to change your source code back to its original state after debugging your program.

2.3 Compiling Programs for Use with Pdbx

Using the **-g** option on the **fortran** or **cc** command line causes the compiler, assembler, and linker to accumulate the debugging information needed by Pdbx. This information includes the following items:

- Name, address, and type of return value (if any) of each procedure
- Names, types, and locations in the stack frame of each procedure's local variables and parameters
- Name, type, and address of each static (global, common, or SAVE) variable
- Address of the first machine instruction resulting from each line of source code
- Name of the source file

The C and FORTRAN compilers store this information in a file with a *.o* suffix. At link time, the linker incorporates the contents of the *.o* file into the executable file, and the *.o* file is deleted.

For example, entering the following command line creates the executable file *fac* for use when debugging the functions **main** and **factorial** in the file *fac.c* (see Figure 1-1 for a listing of *fac.c*).

```
cc -g -o fac fac.c
```

Debugging With Limited Information

Even when a source file is compiled without the **-g** option, the resulting object file contains a symbol table, which includes the names and addresses of all procedures and variables that are accessible from other program modules. This symbol table information is used and updated by the linker and is available to Pdbx unless it is explicitly stripped out of the file. As a result, you can stop a process at the entry to any library routine, including system calls and standard subroutines. You can also step through these routines on the machine instruction level, although you cannot examine the kernel-side code of a system call.

2.4 Linking Programs for Use with Pdbx

When you use the **cc** or **fortran** command, your program is compiled and linked automatically, unless you specified the **–c** option. When you use the **–c** option, you can either reinvoke the C or FORTRAN compiler or explicitly invoke **ld** to create an executable file. When you choose to explicitly invoke **ld**, you must link in the appropriate version of the startup code and run-time libraries and, if you are going to debug your program, you must include the **–lg** option on the **ld** command line to link in Pdbx's run-time library. For example, entering the following commands creates an executable file from the FORTRAN program *fac.f* for use with Pdbx.

```
fortran -g -c fac.f
ld -o fac /lib/crt0.o fac.o -lf -lm -lg -lc
```

Refer to the **fortran**(1) and **cc**(1) man pages for information on specifying the appropriate library files on the **ld** command line.

2.5 Setting Up Your Debugging Environment

You can initialize Pdbx to your specifications using one or more of the following methods:

- *.dbxinit* file
- **–c** option on the Pdbx command line
- Pdbx **source** command

If a file named *.dbxinit* exists in either the current directory or your home directory, the Pdbx commands in that file are executed automatically when Pdbx is invoked. When a *.dbxinit* file exists in *both* the current directory and your home directory, Pdbx uses the *.dbxinit* file in the current directory.

For example, assume that the file *.dbxinit* exists in the current directory. The file contains the following Pdbx command:

```
ignore exit
```

Typically, Pdbx displays the message `Stopped by Exit in _exit` when exiting a procedure in the program being debugged. In this case, Pdbx reads the command to ignore exits in the *.dbxinit* file and does not display the message.

Use the **–c** option to specify a debugger command file for Pdbx to execute on startup. When you use the **–c** option, the commands in the specified command file are executed and all existing *.dbxinit* files are ignored. This option is useful when the standard *.dbxinit* file cannot be used, such as when a *.dbxinit* file contains commands that should not be used when debugging a particular program.

Pdbx reads the *.dbxinit* file or the file specified with the **–c** option before reading the program to be debugged. Thus, these files may contain aliases, **set** commands, and commands to catch or ignore signals, but *not* commands that reference program locations, variables, or affect execution. This includes shell commands or commands such as the following:

```
stop at 20
suspend
trace i
release %5
```

To change your debugging environment *after* invoking Pdbx, use the **source** command. The **source** command executes the Pdbx commands in the specified file as if they were entered at the terminal. Use this command when you find yourself entering the same set of Pdbx commands repeatedly. (Remember, the *.dbxinit* file and the file specified with the **–c** option cannot contain certain types of commands.)

For example, assume you are debugging a C function called **sort** in the file *sort.c*. After invoking Pdbx, you enter the following commands:

```
stop at 35
run
trace i
quit
```

After examining the output from the debugging session, you isolate the bug in another function, modify the code appropriately, recompile, and invoke the debugger to verify that the change fixed the problem. If the previous commands existed in the file *sort.com*, entering the following command would duplicate the debugging session in which those commands were entered manually:

```
source sort.com
```

You can also use the **source** command to repeatedly set and clear the same set of breakpoints and tracepoints at a particular point in a debug session. In the following example, the commands to set breakpoints and tracepoints in the file *setstops.com* aren't executed until after the program has stopped at line number 120.

```
stop at 120
run
source setstops
cont
status
delete 1 2 3
rerun
source setstops
cont
```

For more information on the **source** command, refer to Chapter 5.

2.6 Invoking Pdbx

To debug a program with Pdbx, use the **pdbx** command. The **pdbx** command has the following syntax:

pdbx [*options*] [*filename*]

When specified, *options* must be one or more of the options listed in Table 2-1, and *filename* must be the name of the executable file to debug.

Table 2-1
Pdbx Command Options

Option	Description
–a	Run all processes in the background (asynchronously). Useful when debugging parallel programs. Refer to Chapter 4 for more information on this option.
–C *coredump*	Identify *coredump* as a core file to analyze. Useful when debugging more than one program or more than one core file. Refer to Chapter 5 for more information on this option.

Table 2-1
Pdbx Command Options (cont.)

Option	Description
–c *file*	Execute the Pdbx commands in *file* before reading from standard input. Refer to Section 2.5 for more information on this option.
–I *dir*	Add *dir* to the list of directories that are searched when looking for a source file. Useful when source files are in multiple directories. Refer to Chapters 3 and 5 for more information on this option.
–i	Force Pdbx to prompt for commands (standard input is a terminal). Pdbx defaults to this mode automatically if command input is from a terminal.
–O *execfile*	Identify *execfile* as a program to debug. Useful when debugging applications that consist of multiple programs. Refer to Chapter 5 for more information on this option.
–Q *execfile*	Identify *execfile* as a program that will not be debugged. Useful when debugging applications that consist of multiple programs. Refer to Chapter 5 for more information on this option.
–u	Preserve the case of identifiers for FORTRAN programs. Refer to Chapter 5 for more information on this option.

If you do not specify the name of the executable file, Pdbx uses *a.out*. The following display shows the message Pdbx outputs when invoked.

```
% pdbx fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
```

Note that the header displayed by Pdbx references the debugger as "dbx". You can invoke Pdbx as either **pdbx** or **dbx**: both commands invoke the same program, but each provides a slightly different interface. Invoking Pdbx as **dbx** directs the debugger to ignore exits of child processes and to ignore all execs.

If Pdbx displays the following message when invoked, a core file exists in the current directory:

```
[using memory image in core]
```

As a general rule, you should delete the core file before invoking Pdbx unless you want to examine the status of the program that dumped core. However, once the program in the current executable file is run, you can safely ignore this message. Refer to Chapter 3 for more information on core files. Refer to Chapter 3 for information on using Pdbx *after* it is invoked.

Chapter 3
Using Pdbx

3.1 Introduction 3-1

3.2 Getting Help 3-1

3.3 Error Recovery 3-2

3.4 Language Considerations While Debugging 3-4
Referencing Arrays 3-4
Referencing Pointers, Structs, and Records 3-4

3.5 Referencing Setuid Programs 3-4

3.6 Pdbx Command Line Syntax 3-5

3.7 Common Debugging Tasks 3-6
Creating, Executing, and Deleting Processes 3-6
Exiting Pdbx 3-8
Setting, Displaying, and Deleting Breakpoints 3-8
Displaying the Current Status 3-8
Displaying Source Files 3-9
Displaying and Modifying Variables 3-13
Tracing Program Execution 3-14
Resolving Identifier Conflicts 3-16
Examining the Status of a Program That Dumped Core 3-18
Stopping All Executing Processes 3-18
Determining Where a Process Stopped 3-19
Setting the Radix 3-20
Suspending and Restarting Pdbx 3-21
Executing a Shell Command 3-21
Creating and Deleting Command Abbreviations 3-22
Changing the Context for Interpreting Identifiers 3-22

3.8 Debugging Specialized Applications 3-23
Debugging Applications that Access Physical Address Space 3-23
Debugging Exceptions and Condition Handlers 3-24
Choice of Terminal Signals 3-25
Orphan Processes 3-25
Invoking the Debugger as **dbx** 3-26

Tables

3-1 run vs. create 3-6
3-2 Frequently Used Trace Commands 3-15
3-3 Changing the Radix 3-20

Chapter 3
Using Pdbx

3.1 Introduction

This chapter describes how to use Pdbx *after* you invoke it, and includes step-by-step instructions for performing some common debugging tasks. This chapter does not include information on debugging parallel applications; refer to Chapter 4 for information on this topic.

If this chapter does not describe the debugging task you have in mind, and you don't know the names of the Pdbx commands to use to perform that task, refer to the beginning of Chapter 5 for a list of Pdbx commands classified by function. If you already know the names of the commands to use, but would like additional command-specific information, refer to the description of that command in Chapter 5.

CAUTION

You can crash the system if you single-step your application through an instruction that directly accesses the system's physical address space. Be sure to read Section 3.8.1 before debugging your application with Pdbx when your application directly accesses the system's physical address space (for example, by means of the **pmap**(7) *device driver).*

3.2 Getting Help

If you aren't sure of the command you need *during* a debugging session, use the **help** command. The **help** command displays a list of frequently used Pdbx commands with a brief description of each command's function.

3.3 Error Recovery

Pdbx displays the following types of messages:

- Warning — informational only, no change in the debugging session
- Error — results in Pdbx attempting to recover to a known point in the debugging session, usually the `(pdbx)` command prompt
- Fatal error — results in immediate termination of the current Pdbx session and returns you to the shell from which you invoked Pdbx

If Pdbx displays a message that you do not understand, look up the message in Appendix A (an alphabetized list of common error and warning messages). For each message, a corrective action is recommended.

Pdbx may display the message `syntax error` when it cannot distinguish between a program variable and one of Pdbx's reserved words (typically a command). If Pdbx displays the `syntax error` message in response to a command, reenter the command, this time enclosing the variable in back quotes (`). The following debugging session illustrates the use of back quotes to recover from a syntax error. Note that `all`, `skip`, `step`, and `to` are Pdbx reserved words as well as program variables. (Refer to Appendix B for a list of Pdbx reserved words.)

```
(pdbx) list 1,15 Return
    1           program all
    2   c
    3           integer skip, step,to,i
    4           i =1
    5           skip = 2
    6           step = 2
    7           to = 100
    8           if (i .lt. to) skip = skip * step
    9           end
(pdbx) stop at 8 Return
[1] stop at 8
(pdbx) run Return
  %2  Stopped at breakpoint 1 in all at line 8
    8           if (i .lt. to) skip = skip * step
(pdbx) print to Return
print to
      ^ syntax error
(pdbx) print 'to' Return
100
```

More severe errors can cause processes to abort and dump core. Unless they are caught by a program designed to catch these type of errors, floating-point exceptions, segmentation faults, and attempted divide-by-zero operations always cause a core dump. *Dumping core* means that when the process aborts, the operating system creates a file named *core* that holds the state of the process at the time it aborted. Refer to Section 3.7.9 for information on analyzing core dumps.

3.4 Language Considerations While Debugging

Pdbx uses a combination of Pascal and C syntax when referring to compound data structures. You must use brackets, not parentheses, to specify arrays.

3.4.1 Referencing Arrays

Use the following syntax to specify array elements for FORTRAN and C programs:

array [*index*]

Two-dimensional FORTRAN arrays are specified as follows:

array [*i*,*j*]

Two-dimensional C arrays are specified as follows:

array [*i*] [*j*]

3.4.2 Referencing Pointers, Structs, and Records

For C programs, you can use the following constructs:

- *record*.*field* for records and structs
- *p*^ or **p* to reference the value pointed to by pointer *p*
- *&v* to reference the address of variable *v*

The following constructs are synonymous:

pointer^.*field*
pointer->*field*
pointer.*field*

3.5 Referencing Setuid Programs

A program that runs as another user because the setuid bit is set will not run as another user under Pdbx. Instead, Pdbx runs the program as if it is owned by the current user, which causes the program to eventually fail. When the program you are debugging calls another program whose setuid bit is set, use the **catch fork** and **release** commands. Using these commands allows the program whose setuid bit is set to execute, but precludes debugging of that program. Refer to the description of the **release** command in Chapter 5 for more information.

3.6 Pdbx Command Line Syntax

A Pdbx command line has the following syntax:

cmdname [*parameters*]

cmdname must be a valid Pdbx command and, when specified, *parameters* must be one or more valid parameters for that command. Some commands are referenced by a symbol or a user-specified value rather than a name. These include the following commands:

```
%
/pattern/
?pattern?
address, address/format
address/count format
value=format
```

Refer to the "Commands by Function" listing at the beginning of Chapter 5 for a complete list of Pdbx commands.

Use semicolons to separate multiple commands on the same command line. Pdbx distinguishes between uppercase and lowercase characters—be sure to use lowercase characters for all Pdbx commands.

3.7 Common Debugging Tasks

The remainder of this chapter describes procedures to follow for common debugging tasks. If this section does not describe the debugging task you have in mind, and you don't know the name of the Pdbx command that performs the task, refer to the beginning of Chapter 5 for a list of Pdbx commands classified by function.

3.7.1 Creating, Executing, and Deleting Processes

You can use the **run** or **create** command to create a process. Use the **run** command to terminate any existing processes, to create a new process from the specified executable file, and to begin execution of the new process. Use the **create** command when you do not want to terminate existing processes, as when no current process exists and you want to execute a function defined within the program. Using the **create** command also causes the process to stop before executing any code. You can use the **cont** command to begin execution of a process created by the **create** command. Table 3-1 summarizes the similarities and differences between these two commands.

Table 3-1
run vs. create

Effect on . . .	run	create
new process	creates	creates
current processes	*terminates* existing processes	does *not* terminate existing processes
program execution	*automatically* begins execution of the new process	must enter **cont** command to begin execution of the new process

If you need to pass any command-line arguments to the program, include them on the command line. For example, in the following session, the program is run with 5 as an argument.

```
(pdbx) run 5 [Return]
[Process executes; Pdbx stops it just before it exits.]
5!=1
  %2  Stopped by Exit in _exit at 0x314b
00000314b  exit[]
```

Note that the process you created is %2. Process %1, the program image, is deleted as a result of entering the **run** command.

To rerun the program, use the **run** command again, specifying any arguments on the command line. To run the program again with the *same* arguments, you can use the **rerun** command.

To delete a process, use the **terminate** command. Entering the terminate command deletes the specified process. When you do not specify a process number, the current process is deleted. In the following example, the process associated with process number %3 is deleted. An asterisk (*) marks the current process.

```
(pdbx) ps [Return]
* %2  Stopped at breakpoint 1 in main at line 14
  %3  Stopped by Exit in _exit at 0x1234
  %4  Exited
(pdbx) terminate %3 [Return]
(pdbx)
```

Refer to Chapter 5 for more information on the **run**, **rerun**, **create**, and **terminate** commands.

3.7.2 Exiting Pdbx

Use the **quit** command to exit Pdbx. Entering the quit command also terminates any active processes.

3.7.3 Setting, Displaying, and Deleting Breakpoints

Use the following commands to set, display, and delete breakpoints:

- **stop**—Set a breakpoint at the specified location
- **status**—List the currently defined breakpoints and tracepoints
- **delete**—Cancel the breakpoints or tracepoints as specified by the command number.

In the following example, one of the two existing breakpoints is deleted, and the program is run again. The number in brackets is the *command number*, the number assigned by Pdbx to the command that was used to set the breakpoint. Note that since the breakpoints or tracepoints associated with [1] and [2] were deleted earlier in the session, they are not listed.

```
(pdbx) status Return
[3] stop at 13
[4] stop at 14
(pdbx) delete 4 Return
(pdbx) run 5 Return
  Stopped at breakpoint 1 in main at line 13
   13    n = atoi (argv[1]);
```

3.7.4 Displaying the Current Status

Use the **ps** command to display a list of all processes and each process's state. An asterisk (*) marks the current process. Processes that have been terminated (by means of the **terminate**, **run**, or **rerun** commands) are not listed.

In the following example, process %2 is the current process (as identified by the asterisk).

```
% pdbx fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) stop at 14 Return
[1] stop at 14
(pdbx) run 3 Return
  %2  Stopped at breakpoint 1 in main at line 14
   14    if (n<1 || n>12) {
(pdbx) ps Return
* %2  Stopped at breakpoint 1 in main at line 14
```

3.7.5 Displaying Source Files

Use the **list** command to display the source code for a program. Pdbx displays the first ten lines in the file, starting with the first line. Once you've entered the **list** command, simply enter the **list** command again to display the next ten lines in the file. This time, Pdbx starts listing the source file starting at the last line listed by the previous **list** or search command (*/pattern/* or **?***pattern***?**).

You can change the number of source lines Pdbx displays at one time *around* a procedure (that is, the number of lines before and after a procedure) by setting the predefined debugger variable *$listwindow* to the number of lines you want displayed. When *$listwindow* is set *and* you specify the name of a procedure, Pdbx displays *$listwindow* number of lines. The following session illustrates the use of the **set** and **list** commands. Note that since line 28 is the last line of the file, only 17 lines are displayed.

```
% pdbx fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) set $listwindow=22 Return
(pdbx) list factorial Return
   12
   13        n = atoi (argv[1]);
   14        if (n<1 || n>12) {
   15           printf ("Number out of range.\n");
   16           exit(1);
   17        } else
   18           printf ("%d! = %d\n", n, factorial(n));
   19     }
   20
   21     factorial (n)
   22         int n;
   23     {
   24         if (n = 1)
   25            return (1);
   26         else
   27            return (n * factorial(n-1));
   28     }
```

Refer to the description of the **set** command in Chapter 5 for more information on the *$listwindow* debugger variable.

When the executable file you are debugging was created by compiling more than one source file, use the **file** command to specify which source file to use. You must specify the exact name that was used on the compile command line, since this is the name Pdbx uses to access symbols associated with that file. If you lose track of what file is the current source file, enter the **file** command without parameters. The following session demonstrates the use of the **file** command.

```
% cc -g -o x ../dir1/x.c ../dir2/y.c Return
% pdbx x Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) file Return
../dir1/x.c
(pdbx) file ../dir2/y.c Return
```

If the executable file you are debugging was created by compiling a single source file, the **file** command is needed only to display the source for a file other than the current program.

When the source files for the program being debugged are not in the current directory and the source file pathnames you specified on the compile command line are not absolute pathnames, you may need to use the **use** command or the **–I** option (when invoking Pdbx) to specify additional pathnames. The current source file designation (as displayed by the **file** command) is prefixed by the pathnames specified using the **use** command or the **–I** option so that Pdbx can locate and access the correct source file. Refer to Chapter 5 for more information on the **–I** option.

NOTE

Pdbx does not allow the use of the constructs "~" and "~username" in the directory specification for the **use** *command.*

The following session demonstrates the use of the **use** command to direct Pdbx to the location of a source file.

```
[current directory is /usr/mary/src]
% cc -g -o ../bin/fac fac.c Return
% cd ../bin Return
% pdbx fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) list Return
couldn't read "fac.c"
(pdbx) use ../src Return
(pdbx) list Return
    1   /*
    2    * Usage: fac n
    3    * Computes n factorial (n!) using a recursive algorithm.
    4    * Line 24 contains an intentional error.
    5    */
    6
    7   main (argc, argv)
    8           int     argc;
    9           char    **argv;
   10   {
```

Invoking Pdbx with the **–I** option has the same effect as the **use** command. This is demonstrated by the following session.

```
[current directory is /usr/mary/src]
% cc -g -o ../bin/fac fac.c Return
% cd ../bin Return
% pdbx -I ../src fac Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) list Return
    1   /*
    2    * Usage: fac n
    3    * Computes n factorial (n!) using a recursive algorithm.
    4    * Line 24 contains an intentional error.
    5    */
    6
    7   main (argc, argv)
    8           int     argc;
    9           char    **argv;
   10   {
```

3.7.6 Displaying and Modifying Variables

To display the value of a variable, use the **print** or **dump** command. Use the **print** command to display the value of a *specific* variable, expression, or register. Use the **dump** command to display *all* active variables and their values in the current process. A variable is *active* when the procedure that references it has begun executing but has not returned.

The following example uses the fac function (see Figure 1-1 for a listing of *fac.c*) to illustrate the use of the **dump** command.

```
(pdbx) stop at 14 Return
[1] stop at 14
(pdbx) run 5 Return
  %2 Stopped at breakpoint 1 in main at line 14
   14          if (n<1 || n>12) {
(pdbx) dump Return
main (argc=2, argv = 0x3fffef4c, 0x3fffef58), line 14 in "fac.c"
n = 5
```

In the next example, the value in register eax is displayed. The DYNIX/ptx compilers place the value returned from a function in register eax. Note that the dollar sign character ($) precedes all references to registers, as required by Pdbx.

```
(pdbx) print $eax Return
0x5
```

To alter the value of a program variable, use the **assign** command. In the following example, the value for variable **isvalid** is set to –1 to allow examination of the error processing code in the program.

```
(pdbx) assign isvalid = -1 Return
```

For more information on the **print**, **dump**, or **assign** commands, refer to Chapter 5.

3.7.7 Tracing Program Execution

To print information about a program's state as it executes, use the **trace** command. Entering **trace** without parameters at the Pdbx prompt directs Pdbx to print each line of source code as it executes for the current program and for every procedure it calls.

Two other frequently-used forms of the **trace** command are **trace *procedure*** and **trace *variable***. You can use the **trace *procedure*** command to trace calls to, and returns from, the specified procedure. Each time the procedure is called, Pdbx reports who called it and the values of the parameters, if any. When the specified procedure is a function, Pdbx reports its return value.

Use the **trace *variable*** command to print the value of *variable* each time it changes. Be aware that a **trace** command of this form slows down execution speed. Consider using **trace *variable* at *linenumber*** instead.

Table 3-2 lists and describes some frequently-used versions of the **trace** command.

Table 3-2
Frequently Used Trace Commands

Command	Description
trace	print source code as it executes and all called procedures called by the current program
trace *procedure*	print calls to and returns from the specified procedure
trace *variable*	print specified variable every time it is modified

Use the **status** command to list the current tracepoint and breakpoint settings. Use the **delete** command to remove the settings.

Other forms of the **trace** command exist, but are not explained here. For a complete description of the **trace** command and all its options, refer to Chapter 5.

In the following example, the variable *n* in procedure **factorial** is traced. (See Figure 1-1 for a listing of the source code for this program.)

```
(pdbx) ignore exit Return
(pdbx) trace n Return
[using fac.factorial.n]
[1] trace fac.factorial.n in factorial
(pdbx) run 5 Return
%2  initially (at line 23 in "fac.c"):fac.factorial.n = 5
%2  after line 24 in "fac.c":fac.factorial.n = 1
5! = 1
```

Refer to Section 3.7.3 for information on listing and deleting the current tracepoints. For a complete explanation of tracing options, refer to the description of the **trace** command in Chapter 5.

3.7.8 Resolving Identifier Conflicts

If an identifier appears in more than one context, you may need to specify the context of the identifier explicitly using the syntax *context.identifier*. You must define *context* as one of the following:

- Name of a source file, with the *.c*, *.f*, *.for*, or *.cmp* suffix removed
- Name of a FORTRAN common block
- Name of a procedure, function, subroutine, or program (prefixed with a context specification, if necessary, to prevent ambiguity)

You can use the following Pdbx commands to help you express identifiers in an unambiguous manner:

- **whereis**–lists the fully qualified version of every program element with the specified name
- **which**–reports which of several possible program elements with the specified name is referred to in the current context
- **whatis**–prints the full declaration of the indicated program element

For example, suppose you want to list the first few lines of a procedure named **eval()**. You enter the following command:

```
list eval
```

But Pdbx responds with the following message:

```
"eval" is not a procedure or function
```

Apparently there is another identifier named **eval** in the program. You can use the **which** command to determine what **eval** means in the current context.

```
(pdbx) which eval Return
.eval
```

This means that *eval* is the root of the name of a source file. You can verify this with the **whatis** command.

```
(pdbx) whatis eval Return
source file "eval.c"
```

To find out how to specify the **eval()** procedure instead of the *eval.c* source file, you can use the **whereis** command to list all program elements with the name "eval".

```
(pdbx) whereis eval Return
eval.eval .eval
```

You know that `.eval` represents the source file, so `"eval.eval"` must be the procedure—the procedure named **eval()** in the source file *eval.c*. To list the procedure, you must enter the command **list eval.eval**.

3.7.9 Examining the Status of a Program That Dumped Core

In day-to-day operations, certain error conditions can cause a process to abort and dump core. When the process aborts, the operating system creates a file called *core* that contains the state of the process at the time it aborted. The core file includes information on memory contents, register contents, and pending signals at the time the process aborted. You can use the **ps** and **where** commands to determine the cause of the core dump, as demonstrated in the following example. Entering the **ps** command causes Pdbx to print the cause of the core dump; entering the **where** command causes Pdbx to print where the program aborted.

```
% pdbx fac (Return)
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
[using memory image in corefile]
(pdbx) ps (Return)
* %1  Core in "core" caused by
Segmentation fault in prog.badproc at line 7
(pdbx) where (Return)
prog.badproc(), line 7 in "prog.c"
main(0x1, 0xfff828, 0xfff830), line 11 in "prog.c"
```

Using the **where** command when the current core file was not produced from the executable file associated with the current debugging session produces erroneous information.

3.7.10 Stopping All Executing Processes

Use the **stop all** command to stop all executing processes. When a process is running in the foreground, you must stop that process before entering **stop all**. To stop a foreground process, use the terminal interrupt character (typically Ctrl-C or DEL) or the terminal stop character (typically Ctrl-Z). In the following session, all processes are stopped. Note that Pdbx echoes the signal as ^Z.

```
(pdbx) run 10 [Return]
[Ctrl-Z]
^Z  %2  Stopped by Stop signal from terminal in pfac at line 24
   24   75       continue
(pdbx) stop all [Return]
```

This command is especially useful when debugging programs that generate multiple processes.

3.7.11 Determining Where a Process Stopped

Once a process is stopped, you can use the **ps** command to list the reason the process stopped and the source line at which it was executing when it stopped. You can also use the **where** command to print a traceback from the last procedure invoked to the main program. The traceback includes the name of each procedure, the values of the parameters passed to it, and the source line at which the procedure stopped.

In the following example, entering the **ps** command shows that the current process, %2, is stopped at breakpoint 2 in the procedure **subc**. Entering the **where** command shows the following:

- **subc** was called by **subb**
- **subb** was called by **suba**
- **suba** was called by **check**
- **check** was called by **main**

In the following output from the **where** command, `subc.subc` represents the procedure **subc** in the file *subc.f*. Note that Pdbx qualifies the procedure names since both the procedure name and the source filenames are identical. Pdbx qualifies the procedure names by prefixing the procedure name with the source code filename (minus its *.c* or *.f* suffix). The procedure **main** is automatically included in all FORTRAN programs and is analogous to the C function **main**. For more information on **main**, refer to the *ptx/FORTRAN User's Manual.*

```
(pdbx) ps Return
* %2  Stopped at breakpoint 2 in subc.subc at line 3 in file "subc.f"
(pdbx) where Return
subc.subc(), line 3 in "subc.f"
subb.subb(), line 4 in "subb.f"
suba.suba(), line 4 in "suba.f"
check.check(), line 3 in "check.f"
main.main(0x1, 0x3fffeb14, 0x3fffeb1c) at 0x9361
(pdbx) quit Return
```

For more information on the **ps** and **where** commands, refer to Chapter 5.

3.7.12 Setting the Radix

You can change the radix Pdbx uses to *interpret* command input from decimal to either octal or hexadecimal. You can also change the radix Pdbx uses to *display* values from decimal to hexadecimal only. Use the predefined debugger variables listed in Table 3-3 to set the radix for displaying and interpreting variables.

Table 3-3
Changing the Radix

Variable Type	Debugger Variable			
	octal		hexadecimal	
	display	interpret	display	interpret
character	n/a	n/a	$hexchars	n/a
character array	n/a	n/a	$hexstrings	n/a
floating-point	n/a	n/a	$hexfloats	n/a
integer	n/a	$octin	$hexints	$hexin

3.7.13 Suspending and Restarting Pdbx

Use the **suspend** command to suspend Pdbx and return control to the shell from which you invoked Pdbx. You can use the **fg** command to resume execution of Pdbx. Entering Ctrl-Z will *not* suspend Pdbx.

NOTE

The **suspend** *command is not supported by the Bourne shell. If you are running the Bourne shell, use the* **sh** *command to suspend execution of Pdbx.*

3.7.14 Executing a Shell Command

To execute a shell command during a debugging session, use the **sh** command. The name of the command to execute must follow the **sh** command on the command line.

When Pdbx is invoked from the Bourne shell, you can use the **sh** command to suspend your debugging session. Simply exit the shell to return to your debugging session.

NOTE

Do not use the **sh** *command to execute a program whose setuid bit is set. Refer to Section 3.5 for more information.*

The following example uses the **sh** command to print the current date and time.

```
(pdbx) sh date Return
%1 Tue Mar 14 12:02:14 PST 1989
%2 %2 (pdbx) quit
```

The **sh** command causes Pdbx to create a new shell, execute the command, exit the shell, and prompt for the next debugger command. The value of the **SHELL** environment variable determines which shell program Pdbx uses.

3.7.15 Creating and Deleting Command Abbreviations

You can abbreviate commands and command lines to a specified string using the **alias** command. The **alias** command has the following form:

```
alias alias_name command_string
```

command_string must be quoted if contains spaces.

In the following example, "sx" is created as an alias for the longer string "stop in sub1 if x < 1".

```
(pdbx) alias sx "stop in sub1 if x < 1" Return
```

Several aliases are predefined by Pdbx, but can be redefined. To display a list of current aliases, including those predefined by Pdbx, enter the **alias** command without parameters.

Parameterized aliases are also useful. Refer to the **set** command description in Chapter 5 for more information on creating parameterized aliases.

To delete an alias, use the **unalias** command. For example, entering the following command deletes the alias sx:

```
unalias sx
```

For a complete description of the alias and unalias commands, refer to Chapter 5.

3.7.16 Changing the Context for Interpreting Identifiers

You can use the **down**, **up**, and **func** commands to designate a procedure as the *current procedure*. The current procedure designation provides the context for interpreting identifiers.

The **down** and **up** commands reference procedures in relation to their position on the stack, while the **func** command references procedures by name. As their names suggest, the **down** command moves *down* the stack, toward the currently active procedure, while the **up** command moves *up* the stack, toward the main program. You can specify the number of levels to move up or down the stack as an optional argument; the default value is 1. These commands are especially useful when debugging a recursive procedure.

Use the **func** command to designate the specified procedure as the current procedure. Pdbx interprets identifiers in the scope of the specified procedure and treats the source file containing that procedure as the current source file.

The following example shows the use of the **up** command to identify the function **subb()** as the current procedure.

```
(pdbx) where Return
subc.subc(), line 3 in "subc.f"
subb.subb(), line 4 in "subb.f"
suba.suba(), line 4 in "suba.f"
check.check(), line 3 in "check.f"
main.main(0x1, 0x3fffeaf4, 0x3fffeafc) at 0x9361
(pdbx) up Return
subb.subb(), line 4 in "subb.f"
(pdbx) whatis subb Return
subroutine subb()
```

3.8 Debugging Specialized Applications

This section describes applications that have "special" debugging requirements. It does *not* include information on debugging parallel programs. Refer to Chapter 4 for information on debugging parallel programs.

3.8.1 Debugging Applications that Access Physical Address Space

To examine or modify the contents of a process's address space, Pdbx uses the **ptrace()** system call. This means that Pdbx is executing *kernel* code, not *user* code, when it examines or modifies an address location in a process being debugged. In kernel code, stray memory references such as references to nonexistent memory or unresponsive memory-mapped devices are not expected, and typically cause the system to panic (crash).

Using **ptrace()**, Pdbx can catch stray memory references and prevent system panics and security breaches. However, if you single-step your application across an instruction that directly accesses the system's physical address space (as when reading or writing a special memory-mapped device), you can cause a system crash. Therefore, use extreme caution when debugging applications that access the system's physical address space.

3.8.2 Debugging Exceptions and Condition Handlers

Read this section if your program explicitly catches or ignores one or more signals.

A *signal* is an event generated by software and is analogous to a hardware interrupt. A process may receive a signal due to an event on the process's terminal, (as when entering Ctrl-Z or turning off the terminal), or as a result of an internal exception (for example, division by zero). A process may also receive a signal from another process. The receiving process may be set to *catch* the signal (that is, deal with the signal with a signal handler routine), *block* the signal until a later time, or *ignore* (discard) the signal. When the process has not been set to catch, block, or ignore the signal, the process stops or aborts. Signals are described in **signal**(2).

Entering the **catch** and **ignore** commands without arguments lists which signals are currently caught or ignored by Pdbx.

A process being debugged can never receive a signal without first stopping and notifying Pdbx. When Pdbx sees that the process has been sent a signal, it takes one of two courses of action.

1. If Pdbx has been directed to catch the signal, Pdbx displays a message that the signal has been received and prompts for a command. If program execution is subsequently resumed (with the **cont** command for example), Pdbx delivers the signal to the process unless the **cont** command explicitly suppresses delivery of the signal.
2. If Pdbx has been directed to ignore the signal, Pdbx delivers the signal to the process. The effect on the process depends on the type of signal and whether the process is set to handle, block, or ignore the signal.

For example, suppose that process %5 is running in the foreground and you type the terminal interrupt character (typically Ctrl-C or DEL). Here's what happens:

1. Typing Ctrl-C sends a `SIGINT` signal to process %5.
2. Since the process is under Pdbx control, the process is stopped and Pdbx is notified that %5 has received a `SIGINT` signal.
3. By default, Pdbx catches `SIGINT` signals, so Pdbx displays a message that the signal has been received and prompts for a command.
4. If you type **cont**, Pdbx delivers the signal and, unless the process is set to handle, block, or ignore `SIGINT` signals, the process exits. If you type **cont 0** instead, Pdbx discards the signal and resumes execution of the process as if it had never stopped.

Processes that are stopped or running in the background can be sent signals using the **signal** command. For example, suppose %5 is stopped and Pdbx is set to catch SIGINTs. You type:

```
(pdbx) signal %5 sigint
```

Here's what happens:

1. Pdbx sends a SIGINT to %5 using the **kill()** system call.
2. When you next resume execution of %5 the process is immediately stopped and Pdbx is notified that %5 has been sent a SIGINT. Since Pdbx is catching SIGINT signals, Pdbx announces the signal and prompts for a command.
3. As before, typing **cont** causes Pdbx to deliver the signal, and the process exits. Typing **cont 0** causes Pdbx to discard the signal and resume execution of the process.

Choice of Terminal Signals

By convention, if you continue a process with a stop signal (e.g., SIGSTOP or SIGTSTP), the stop signal is discarded by the debugger. Therefore, if you have stopped a process with Ctrl-Z (SIGTSTP), you do not need to explicitly suppress delivery of SIGTSTP when you continue execution of the process. (That is, **cont** is equivalent to **cont 0** in this case.) On the other hand, if you stop a process with Ctrl-C (SIGINT), you must use **cont 0** to suppress delivery of the SIGINT, since by default SIGINT aborts the process. For this reason, you probably want to use Ctrl-Z rather than Ctrl-C to stop a process running in the foreground.

Orphan Processes

Note that if your program handles or ignores signals from the terminal and you direct Pdbx to ignore such signals, you can find yourself in a situation where the only way to stop a foreground process is to use the DYNIX/ptx **kill -9** command from another terminal. To avoid this situation, Sequent recommends that you do one of the following:

- allow Pdbx to catch at least one type of terminal signal
- allow your program to be stopped or aborted by at least one type of terminal signal

For more information about signal handling in Pdbx, refer to the descriptions of the **catch**, **ignore**, **cont**, and **signal** commands in Chapter 5.

Invoking the Debugger as dbx

When invoked as **dbx**, the debugger ignores exits of child processes and execs of all processes. Invoking the debugger as **dbx** is useful when you are debugging a program that creates multiple processes and only the parent process is of interest.

Chapter 4
Debugging Parallel Applications

4.1 Introduction 4-1

4.2 Considerations for Parallel Applications 4-2
Debugger Context 4-2
Current Process and Current Program 4-2
Current Source File and Current Source Line 4-3
Current Procedure 4-3
Running Processes Asynchronously 4-4
Handling Signals 4-5
C$DOACROSS Loops 4-6

4.3 Debugging Heterogeneous Applications 4-6

Tables

4-1 Comparing C Shell and Pdbx Commands for Terminal I/O 4-4

Chapter 4
Debugging Parallel Applications

4.1 Introduction

You can use Pdbx to debug parallel applications. A *parallel application* consists of two or more cooperating processes that execute concurrently. The simplest kind of parallel application, sometimes referred to as a *homogeneous application*, includes a parent process and one or more child processes all running the same program. When you debug such an application, Pdbx creates the parent process and the parent subsequently creates its children using the **fork()** system call or the **m_fork()** Parallel Programming Library routine.

Pdbx is also useful in debugging applications that consist of multiple programs, or heterogeneous applications. A *heterogeneous application* consists of a single process that executes one program and then another, or it consists of multiple, possibly unrelated, processes running different programs. A client/server application is an example of the latter type.

When debugging parallel applications, ordinary debugging operations such as setting breakpoints, starting program execution, and providing program input from the keyboard may take on new dimensions. In fact, some Pdbx commands and command options apply only to multiple process environments. For example, consider the following points when using the **stop** command to set breakpoints in a homogeneous parallel application:

- By default, each breakpoint is set in all processes running the same program, and when a breakpoint is encountered, only the processes that encounter the breakpoint are stopped. Processes that do not encounter the breakpoint continue executing.
- You can specify that a **stop** command applies only to a particular process. In this case, only the specified process is stopped.
- Using the **all** parameter of the **stop** command, you can direct Pdbx to stop all processes when any process hits the breakpoint. The process that encounters the breakpoint is stopped immediately; Pdbx stops the remaining processes with a signal as soon as possible.

- The current implementation does not support stopping **all** processes when a specified process hits a breakpoint.
- Previously set breakpoints are inherited by new processes as they are created.

This chapter provides information that is useful when debugging parallel applications. Section 4.2 describes debugging implications for both types of parallel applications. Section 4.3 describes debugging implications for heterogeneous applications only.

4.2 Considerations for Parallel Applications

The following list describes factors to consider when debugging parallel applications:

- Current debugger context
- Asynchronous execution of processes
- Signals
- Pdbx's interpretation of C$DOACROSS loops

4.2.1 Debugger Context

Pdbx interprets commands in relation to the current debugger context. The following sections describe aspects of the debugger context that are important in a multiple process environment.

Current Process and Current Program

Typically, the *current process* is the process created most recently by the **run**, **rerun**, or **create** command. In the display produced by the Pdbx **ps** command, the current process is marked with an asterisk (*). The current process designation can be explicitly changed using the % command.

By default, the current process is used by commands that control process execution (such as **cont** or **step**) and commands that access a process's variables (such as **assign** or **print**). For a command such as **cont**, you can designate a different process *n* by including a parameter after the command name. In commands that fetch the values of a process's variables, you can prefix an identifier or expression with "*n*:" to indicate that the value is to be fetched from process *n*.

The *current program* is the program being executed by the current process. The current program is used by commands such as **file** and **use**, which identify the source file and define the current source file.

Current Source File and Current Source Line

Pdbx uses the *current source file* when interpreting line number references and when listing source code with the **list** command. When you enter the **list** command without parameters, Pdbx lists source code starting at the *current source line* in the current source file. Traces and stops are also set in the program associated with the current source file (not necessarily the current program).

Each process has its own current source file and current source line. You can explicitly change a process's current source file using the **file** command. The current source file is also changed as a side-effect of using commands such as **func** and **list**. For example, a command of the form **list** ***proc*** changes the current source file designation to the file that contains procedure *proc*. Also, when a process stops, its current source file and current source line designations are updated so that subsequent **list** commands list the source lines following the last line displayed (that is, the next line to be executed).

Current Procedure

The current procedure designation provides the context for interpreting identifiers. By default, the *current procedure* is the one that was executing when the process was stopped. The current procedure designation can be changed explicitly using the **func**, **up**, and **down** commands. When more than one instance of a recursive procedure is active, you can use the **up** and **down** commands to move the current procedure designation to the selected instance. The current procedure designation is maintained separately for each process.

Refer to Chapter 3 for more information on the **func**, **up**, and **down** commands.

4.2.2 Running Processes Asynchronously

By default, Pdbx uses synchronous terminal I/O. That is, while the current process is running, all terminal I/O is associated with that process. Pdbx does not prompt for a command until the process is stopped by a breakpoint, signal, or other event.

When debugging parallel applications, it may be useful to enter commands while one or more processes are running. Therefore, Pdbx commands that put a process into execution (**run**, **rerun**, **cont**, **next**, **step**) accept an optional **&** parameter, which indicates that the process is to run in the background. After starting the process running, Pdbx immediately prompts for another command. If a process running in the background needs to read from the terminal, Pdbx stops and displays the following message:

```
stopped (tty input)
```

For that read to complete, you must bring it into the foreground by using the **cont** command.

You can use the **–a** option when invoking Pdbx to specify that all processes run in the background; this option removes the need to append the **&** to commands like **cont**. Don't use the **–a** option if any process will need to read from the terminal.

Pdbx's handling of asynchronous terminal I/O is similar to that of the C shell. C shell users may find Table 4-1 helpful.

Table 4-1
Comparing C Shell and Pdbx Commands for Terminal I/O

C Shell Command	Pdbx Command	Action
cmd cmdargs &	run *cmdargs* &	Run process in background
CTRL-Z	CTRL-Z	Stop process (but not Pdbx)
fg	cont	Continue process in foreground [a]
bg	cont &	Continue process in background
suspend	suspend	Suspend csh; suspend Pdbx

[a] With Pdbx, if a process is running in the background, it must be stopped using a command such as **stop all** or **signal** before it can be continued in the foreground.

Terminal input is always directed either to the process running in the foreground (if any) or to Pdbx. *Terminal output* from multiple processes (as well as Pdbx messages associated with those processes) may all be sent to the terminal at the same time.

Pdbx interprets terminal signals such as Ctrl-C and Ctrl-Z as directed at the process running in the foreground. If there is no such process running, such signals have no effect. As a result, if you want to suspend Pdbx and return control to the shell that invoked it, you must use the Pdbx **suspend** command; Ctrl-Z will not suspend Pdbx. For more information on signals, refer to the following section.

4.2.3 Handling Signals

Programs linked to the Parallel Programming Library use `SIGSEGV` to grow stack and data space in shared memory. Therefore, for most parallel programs, Pdbx should ignore `SIGSEGV` signals. If you are debugging programs that are linked to the Parallel Programming Library and contain shared variables, Pdbx automatically ignores `SIGSEGV` signals and displays the following message:

```
found symbol _init_shbrk, assuming parallel library used, ignoring SEGV]
```

However, if you suspect your program contains a bad pointer that is causing a true segmentation fault, you should direct Pdbx to catch `SIGSEGV` signals. If the fault occurs before the parallel code is executed, Pdbx will catch the signal.

You can also use Pdbx to monitor the creation of new processes. Using the **catch fork** command, you can direct Pdbx to stop the child process and prompt for a command immediately after executing **fork** or **m_fork**. (The parent is not stopped.)

You can also use Pdbx to monitor various processes as they enter and exit standard synchronization routines such as **s_lock()**, **s_unlock()**, and **s_wait_barrier()**. These and other routines from the Parallel Programming Library are described in the **3PPS** man pages.

If you invoke Pdbx as **dbx**, Pdbx ignores exits of children of the process being debugged and all execs. Refer to Appendix C for more information.

4.2.4 C$DOACROSS Loops

The FORTRAN compiler generates a separate subroutine for each C$DOACROSS loop in a program compiled with the **–mp** option. These subroutines are named in the following manner:

do.*label*.*index_variable_name*.*procedure_name*

or

do.*line_number*.*index_variable_name*.*procedure_name*

(You can display the names of all subroutines generated for C$DOACROSS loops by generating a listing file with the **–Vm** compiler option.)

When debugging a parallel program, you should direct Pdbx to ignore exits **or** run processes in the background. If you do not, the program appears to be hung. This is because the foreground process is associated with the parent process, and the parent process is waiting for the child (which is stopped) to exit.

A variable local to a FORTRAN parallel loop is different from a variable with the same name outside of the loop. For example, if you trace a variable that is first used as an index variable in a parallel loop and that same variable is also used in another loop, the trace displays only the variable in one of the loops.

When stepping or tracing a FORTRAN program with a parallel loop, Pdbx will not step into or trace the loop because it does not have access to the Parallel Programming Library routine that starts up parallel loops. You can use the **stop** command to set a breakpoint at the first line of the body of the loop, and then enter **cont** until that breakpoint is encountered. After the process stops inside the loop, you can use the **step**, **next**, or **trace** commands. (Using the **cont to *line_number*** command has the same effect as **stop at *line_number*; cont**.)

4.3 Debugging Heterogeneous Applications

When you invoke Pdbx to debug a heterogeneous application, you must use the **–O** option to specify the executable file of every program to debug. Pdbx creates one process from each executable file specified on the **pdbx** command line.

For each program being debugged, Pdbx maintains a separate table of debugging information, a separate current source file designation, a separate list of directories to search for source files, and a separate set of **stop** and **trace** commands. When a process execs, it ignores the **stop/trace** commands defined for its old program and uses any **stop/trace** commands specified for its new program.

You can use the **-Q** option instead of the **-O** option to save memory space and initialization time when invoking Pdbx to debug a heterogeneous application. The **-Q** option is used to specify the program modules that will *not* be debugged and do *not* require Pdbx debugging information. The result of the **-Q** option on a program module is the same as if the program were compiled without the **-g** compiler option. (Refer to Chapter 2 for more information on debugging programs compiled without the **-g** option.)

Chapter 5
Command Dictionary

5.1 Introduction 5-1

5.2 The pdbx Command 5-1

5.3 Pdbx Command Line Syntax 5-3
Commands 5-3
Command Line Format 5-3
Abbreviating Commands 5-4
Case Sensitivity 5-4
Identifiers 5-4
Identifiers Containing Periods 5-4
Identifiers vs. Pdbx Reserved Words 5-5
Qualifying Identifiers 5-5
Line Numbers 5-6
Expressions 5-6
Accessing Values from Different Processes 5-6
Arithmetic and Boolean Expressions 5-7
Addresses 5-7
Hexadecimal and Octal 5-8
Character Strings 5-8
Processor Registers 5-8
Type Conversion 5-9

5.4 Command Descriptions 5-10

Tables

5-1 Examples of Qualified Identifiers 5-5
5-2 Arithmetic and Boolean Operators 5-7
5-3 Symmetry Register Names 5-9
5-4 Format Letters to Use When Displaying Memory 5-15
5-5 Pdbx-Defined Debugger Variables 5-58

Commands by Function

Creating and Executing Processes

%n — Designate current process 16
call — Execute procedure 22
cont — Resume program execution 26
create — Create new process 28
next — Execute next line or procedure 39
ps — List processes being debugged 49
release — Release a process stopped by fork 52
return — Return to procedure 54
run, rerun — Run program from beginning 55
step — Execute one source line 65
terminate — Terminate process(es) 72

Tracing, Breakpoints, and Signals

catch — Break when signal or event occurs 24
delete — Cancel stops or traces 29
ignore — Ignore signal or event 37
signal — Send signal to process(es) 61
status — Print active stop and trace commands 64
stop — Set breakpoint 66
trace — Trace program execution 73

Examining and Altering Variables

assign — Assign value to variable 19
dump — Print variables in specified procedure 32
print — Print value of variable or expression 40

Examining the Current Context

down — Designate called procedure as current procedure 30
func — Designate current procedure 35
up — Designate calling procedure as current procedure 78
whatis — Print declaration for identifier 80
where — Print traceback from current procedure 81
whereis — List all contexts for identifier 82
which — Clarify scope of identifier 83

Accessing Source Files

/ and ? — Search for pattern in source file....12
edit — Edit source file....33
file — Designate current source file....34
list — List part of source file....38
use — Specify directories that contain source files....79

Command Aliases and Debugger Variables

alias, unalias — Define alias for Pdbx command....17
set, unset — Assign value to debugger variable....56

Machine-Level Debugging

address/ and address= — Print contents of memory....14
listi — List disassembled machine code....38
nexti — Execute next instruction or procedure....39
stepi — Execute one machine instruction....65
stopi — Set breakpoint....66
tracei — Trace program execution....73

Miscellaneous

help — Print synopsis of Pdbx commands....36
quit — Exit Pdbx....51
sh — Execute DYNIX/ptx command....60
source — Execute Pdbx commands from a file....63
suspend — Suspend Pdbx and return to shell....71

Chapter 5
Command Dictionary

5.1 Introduction

This chapter contains reference information for the debugger and includes information on debugger invocation, Pdbx's command line syntax, and detailed descriptions of all Pdbx commands.

5.2 The pdbx Command

To invoke the debugger, use one of the following forms of the **pdbx** command. Use the first form when debugging an application that consists of one or more processes running the same program. Use the second form when debugging applications that consist of multiple programs, such as client/server applications. In the latter case, each **–O** option specifies a program to debug.

pdbx [**–i**] [**–u**] [**–a**] [**–c** *file*] [**–I** *dir*] ... [*execfile* [*coredump*]]

pdbx [**–i**] [**–u**] [**–a**] [**–c** *file*] [[**–I** *dir*] ... **–O** *execfile*] ... [**–Q** *execfile*] ... [**–C** *coredump*] ...

You can invoke the debugger as either **pdbx** or **dbx**: both names are links to the same program. However, when Pdbx is invoked as **dbx**, the debugger ignores exits of child processes and all execs.

When Pdbx starts, it flushes any characters typed ahead. Enter Pdbx commands only after the `(pdbx)` prompt is displayed.

The specified *execfile* is produced by compiling a C or FORTRAN program with the **–g** option. You can debug a program compiled without the **–g** option at the machine-level only.

If a file named *core* exists in the current directory or you specify one or more core dump files on the command line, you can examine the state of the program that produced the core dump when it faulted. For more information about analyzing core dumps, refer to Chapter 3.

The command line options are defined as follows:

–a Run all processes in the background (asynchronously). As soon as Pdbx starts a process running, it prompts immediately for another command. No process can read from the terminal.

–C *coredump* Identify *coredump* as a core file to analyze. Use the **–C** option when there is more than one program or more than one core file. The **–O** option must precede the **–C** option on the command line. Pdbx matches each core file with the appropriate executable file.

–c *file* Execute the Pdbx commands in *file* before reading from standard input. Pdbx executes the commands in *file* in place of the commands in *.dbxinit*. Refer to Chapter 2 for more information on initializing Pdbx.

–I *dir* Add *dir* to the list of directories Pdbx searches when looking for a source file. Typically, Pdbx looks for source files first in the directory where *execfile* is located, and then in the current directory. Directories specified using **–I** are added to the search list after the directory that contains *execfile*.

The directory search path can also be modified with the Pdbx **use** command.

When you specify more than one executable file, each **–I** option applies to the program specified by the next **–O** option. For example, the following command invokes Pdbx to debug programs *prog1* and *prog2*. The directory search path for *prog1* is the current directory, *dir1*, and *dir2*, and the directory search path for *prog2* is the current directory, *otherdir*, and *dir3*.

```
pdbx -I dir1 -I dir2 -O prog1 -I dir3 -O otherdir/prog2
```

–i Cause Pdbx to prompt for commands (even when standard input is not a terminal).

–O *execfile* Identify *execfile* as a program to debug. You must specify the **–O** option before the **–C** option on the command line. Do not use this option if you are debugging only one program.

–Q *execfile* Identify *execfile* as a program that should not be debugged. The **–Q** option is similar to the **–O** option described previously, except that it specifies a program that has minimal debugging information when loaded into Pdbx. (Refer to Chapter 3 for more information on files with minimal debugging information.) Use this option when debugging large programs and available memory space is minimal. You must specify the **–Q** option before the **–C** option on the command line.

–u Preserve the case of identifiers for FORTRAN programs. The C compiler always preserves the case of identifiers. The FORTRAN compiler converts all identifiers to lowercase, unless the **–case** option is specified. If the FORTRAN compiler was invoked with the **–case** option, you should invoke Pdbx with the **–u** option to preserve case sensitivity.

5.3 Pdbx Command Line Syntax

This section provides detailed descriptions of all Pdbx command-line elements. Use this section as a reference for the correct syntax of commands, identifiers, line numbers, and expressions on the Pdbx command line.

5.3.1 Commands

When entering Pdbx commands, you should be aware of the following three factors that affect how commands are interpreted by Pdbx:

- Command-line format
- Command abbreviations
- Case sensitivity

The following paragraphs describe each of these factors in detail.

Command Line Format

Pdbx reads a line of text from the terminal, interprets it as a command or series of commands, and executes the indicated commands. Use a semicolon (;) to separate multiple commands in a command line. If Pdbx detects an error in any command on a command line, all subsequent commands are ignored.

A single command can extend over multiple lines; a newline preceded by a backslash is interpreted as a space.

You can insert or omit spaces anywhere in a command, except where doing so would obscure the meaning of the command.

Abbreviating Commands

You can use the Pdbx **alias** command to define abbreviations for commonly used Pdbx command names and command lines. Several aliases are predefined by Pdbx such as **p** as an alias for **print**. To list the currently defined aliases, enter the **alias** command without parameters. Refer to Chapter 3 for more information on using aliases.

Case Sensitivity

Except where otherwise noted, Pdbx distinguishes between uppercase and lowercase letters. For example, **ps** is a valid Pdbx command, but **PS** is not. In general, use uppercase only for character strings and identifiers that include uppercase letters. If you specify the **–u** option when invoking Pdbx, the case of identifiers for FORTRAN programs is preserved. The C compiler always preserves the case of identifiers. The FORTRAN compiler converts all identifiers to lowercase, unless either the **–case** or **–commoncase** option is specified. If the FORTRAN compiler was invoked with the either of these options, invoke Pdbx with the **–u** option to preserve case sensitivity.

5.3.2 Identifiers

Pdbx accepts any identifier accepted by the C or FORTRAN compilers.

As mentioned in the previous section, the case of identifiers at debug time is determined by the compiler and the command line options used during compilation.

Identifiers Containing Periods

When entering an identifier containing a period (.), you must enclose the identifier in back quotes. This forces Pdbx to recognize the period as part of the identifier. For example, entering the following command directs Pdbx to stop in the routine .SQRT.

```
stop in `.SQRT`
```

Identifiers vs. Pdbx Reserved Words

If an identifier in your program is also a Pdbx reserved word, you can enclose the identifier in back quotes to force Pdbx to recognize the word as an identifier. Refer to Chapter 3 for more information.

Qualifying Identifiers

When an identifier appears in more than one context (for example, variables named *i* in two different procedures), you can explicitly specify the context of the identifier using the syntax *context*.*identifier*. *context* can be any of the following items:

- Name of a source file, with the *.c* or *.f* suffix removed
- Name of a FORTRAN common block
- Name of a procedure, function, subroutine, or program (prefixed with a context specification, if necessary, to prevent ambiguity)

Table 5-1 provides examples of qualified identifiers for ambiguous program elements.

Table 5-1
Examples of Qualified Identifiers

Program Element	Qualified Identifier
Variable *v* in function **f**	f.v
Variable *v* in function **f** in source file *subs.c*	subs.f.v
Variable *v* in common block **c**	c.v

NOTE

Use the Pdbx **which**, **whatis**, *and* **whereis** *commands to resolve problems with identifiers used in multiple contexts.*

5.3.3 Line Numbers

A line number in a Pdbx command must have one of the following forms:

- An integer constant — for example, the command **list 25,44** displays lines 25 through 44 in the current source file
- *"filename"*:*n* — for example, the command **stop at "mainline.f":14** stops the program at line 14 in source file *mainline.f*

5.3.4 Expressions

Pdbx recognizes most types of expressions accepted by the C and FORTRAN compilers. However, Pdbx does not recognize FORTRAN `LOGICAL` expressions in Pdbx commands.

Accessing Values from Different Processes

By default, expressions are evaluated in the context of the current process. However, a construct of the following form specifies an expression that is evaluated in the context of process number *n* instead of in the current process. (The Pdbx **ps** command displays process numbers and the **%** command designates the current process.)

n:*expression*

For example, **3:errflag** yields the value of variable *errflag* in process 3, and **5:(a+b)** yields the sum of variables *a* and *b*, both in process 5.

NOTE

A process must be stopped before Pdbx can fetch the value of a variable or register in that process. Thus, entering a command having the form **stop %2 in *procname* if 3:*varname*=1** *produces an error if process 3 is running when process 2 enters procname.*

Arithmetic and Boolean Expressions

On the Pdbx command line, Pdbx accepts the arithmetic and Boolean operators shown in Table 5-2. Pdbx cannot accept Boolean expressions that include `LOGICAL` variables, and issues the following error message:

```
found xxx, expected boolean expression
```

Table 5-2
Arithmetic and Boolean Operators

Operation	Operator
Unary plus	`+x`
Unary minus	`-x`
Add	`x+y`
Subtract	`x-y`
Multiply	`x*y`
Real divide	`x/y`
Integer divide	`i div j`
Modulus	`i mod j`
Logical AND	`boolx and booly`
Logical OR	`boolx or booly`
Comparison	`<, <=, >, >=`
	`=` *or* `==`
	`<>` *or* `!=`

Addresses

An address can take any of the following forms:

- An integer constant — using the prefix `0x` specifies a hexadecimal address
- *&variable* or *&procedure* — yields the address of the specified variable or procedure
- *address+address*, *address-address*, or *address*address*
- **address* — yields the value that the pointer at *address* points to

Hexadecimal and Octal

You must express hexadecimal and octal numbers using C syntax: hexadecimal numbers take the prefix `0x` and octal numbers take the prefix `0` (zero). Decimal numbers have no prefix. Legal hexadecimal digits are 0–9 and a–f or 0–9 and A–F. Legal octal digits are 0–7.

The command **set $hexin** directs Pdbx to interpret all integers in Pdbx commands, with or without the `0x` prefix, as hexadecimal. In the following example, process 16 becomes the current process.

```
(pdbx) set $hexin Return
(pdbx) %10 Return
```

Similarly, **set $octin** directs Pdbx to interpret integers as octal. If both are set, Pdbx interprets integers as hexadecimal. Refer to Chapter 3 for more information on setting the radix for interpreting and displaying variables.

Character Strings

Character strings must be enclosed in single quotes (*'string'*) or double quotes (*"string"*).

Processor Registers

The construct *$regname* yields the current contents of the specified processor register. You must specify the `$` for Pdbx to recognize the string as a register name. Table 5-3 lists registers that Pdbx can display.

Table 5-3
Symmetry Register Names

Register Type	Processor		
	80386	80387	1167
scratch	$eax $ecx $edx	n/a n/a n/a	$fp2–$fp7
user[a]	$ebx $edi $esi	n/a n/a n/a	$fp8–$fp31
stack	n/a	$st0–$st7	n/a
other	$eip $esp $eflags $epb	n/a n/a n/a n/a	n/a n/a n/a n/a

[a]program variable declared with the `register` keyword

The following aliases are also supported: `pc` for `eip`, `sp` for `esp`, and `fp` for `ebp`.

Type Conversion

To coerce the value of expression *expr* to type *typename*, use the construct *expr**typename*. To coerce the value of expression *expr* to a pointer to type *typename*, use the construct *expr*\&*typename*. If *typename* is a struct, precede it with a double dollar sign ($$).

For example, to print the contents of the variable of type `struct node` at address 0xfff000, enter the following command:

print *(0xfff000\&$$node)

You can also use the **set $nostrict** command to relax type-checking rules, as demonstrated by the following example:

```
(pdbx) list 1,22 (Return)
    1   struct test {
    2       int q;
    3       char *s;
    4   };
    5
    6   int prstruct(p)
    7       struct test *p;
    8   {
    9       printf("p->q => %d,p->s => %s\n",p->q,p->s);
   10       return 1;
   11   }
   12
   13   main()
   14   {
   15       struct test demo;
   16
   17       demo.q = 23;
   18       demo.s = "Hi there!!";
   19
   20       printf("Exiting main\n");
   21       exit (0);
   22   }
(pdbx) stop at 20 (Return)
[1] stop at 20
(pdbx) run (Return)
 %2  Stopped at breakpoint 1 in main at line 20
   20       printf("Exiting main");
(pdbx) print demo (Return)
(q = 23, s = "Hi there!!")
(pdbx) print &demo (Return)
0x3fffef14
(pdbx) print prstruct(0x3fffef14) (Return)
type mismatch for p in call to prstruct
(pdbx) set $nostrict (Return)
(pdbx) print prstruct(0x3fffef14) (Return)
p->q => 23,p->s => Hi there!!
1
```

5.4 Command Descriptions

The remainder of the chapter describes all Pdbx commands. Commands are listed in alphabetical order and each command description begins on a new page. Commands that cannot be alphabetized are listed first.

/pattern [/]
or
?*pattern* [?]

pattern The character string to find. Can be any regular expression defined in **ed**(1).

DESCRIPTION

A command of this form directs Pdbx to print the first line in the current source file that contains the specified pattern. The pattern search begins at the line most recently printed or defaults to line 1.

When you specify the pattern with one or two slashes (/), Pdbx searches forward; when you specify the pattern with one or two question marks (?), Pdbx searches backward. If the pattern is not found in the remainder of the file, searching wraps around so that the entire file is searched.

If you do not specify *pattern*, Pdbx use the pattern most recently specified. For example, entering **?** or **??** causes Pdbx to search backward for the next occurrence of the previously specified pattern.

If you enter the **list** command without parameters, and a search command was specified earlier in the session, Pdbx lists source lines starting at the line following the last line displayed with the search command.

EXAMPLES

The following examples use the FORTRAN program *pfac.f* listed in Chapter 1. Assume you are at the beginning of the file.

```
(pdbx) /argbuf/ Return
3      character*8 argbuf
```
(*first occurrence of argbuf found on line 3*)

```
(pdbx) ?argbuf? Return
9       read (argbuf, 50) n
```
(searches back from line 3 and finds it again in line 9 after wrapping around)

```
(pdbx) ? Return
8       call getarg (2, argbuf)
```
(searches backward again using previous pattern and finds it again in line 8)

address ,*address* / [*format*]
or
address / [*count*] [*format*]
or
address = [*format*]

address	Any valid address expression as defined in Section 5.3. *address* is the memory location of the object to print.
format	A format letter specifying the data type of the object to print. Table 5-4 lists the valid format letters and their default radix. The value of *format* defaults to the previously specified format. The initial format is **x**.
count	An integer value that specifies the number of objects to print. Defaults to **1**.

DESCRIPTION

A command of the form *address/* or *address* ,*address/* prints the contents of the specified region of memory. Note that you must include the slash character (/) after the address. The first form specifies a starting address, and the second form defines a range of addresses. For example, the command **&myproc/10i** disassembles and prints the first 10 instructions in procedure **myproc()**, and **./20** disassembles and prints the next 20 instructions.

A command of the form *address=* displays the value of the specified address. For example, the command **&myproc=X** displays the address of **myproc()** in hexadecimal, and **0xf0=d** displays the decimal equivalent of the hexadecimal number f0.

A period (.) specifies the address following the one most recently printed.

Table 5-4 lists the memory formats you can use to display values in memory.

Table 5-4
Format Letters to Use When Displaying Memory

Format	Object Type[a]	Radix
i	machine instruction	n/a
d	short word	decimal
D	long word	decimal
o	short word	octal
O	long word	octal
x	short word	hexadecimal
X	long word	hexadecimal
b	byte	octal
c	character	n/a
s	null-terminated string	n/a
f	single-precision real	n/a
g	double-precision real	n/a

[a] A short word is 16 bits; a long word is 32 bits

EXAMPLES

The following example uses the command **&ptr/X** to display the value of *ptr*.

```
(pdbx) list Return
    1   main()
    2   {
    3     char *ptr;
    4     ptr = (char *) 0x123245;
    5   }
(pdbx) stop at 5 Return
[1] stop at 5
(pdbx) run Return
  %2  Stopped at breakpoint 1 in main.main at line 5
    5   }
(pdbx) &ptr/X Return
3fffef20:  00123245
```

```
%n
```

n The process number, as displayed by the Pdbx **ps** command.

DESCRIPTION

This command makes process *n* the current process. Pdbx changes the following designations to reflect *n* as the current process:

- Current procedure – as specified by the **func** command
- Current source file – as specified by the **file** command
- Source file directory list – as specified by the **use** command

For more information on the current process, current procedure, current source file, or source file directory list, refer to Chapter 3.

```
alias name cmdname
   or
alias name "cmdstring"
   or
alias name ( parameter [ ,parameter ] ... ) "cmdstring"
   or
alias [name]
   or
unalias name
```

name	The name of the alias (any identifier).
cmdname	A Pdbx command.
cmdstring	A string of characters to use as command input when the alias is entered as a command. The string can contain multiple commands separated by semicolons (;). You must precede any double quote or backslash within the string with a backslash (\" or \\). A newline preceded by a backslash is treated as a space.
parameter	The name of a parameter referenced in *cmdstring* (any identifier).

DESCRIPTION

The **alias** command defines an abbreviation for a Pdbx command or a Pdbx command string. After creating an alias, you can use the alias instead of the longer command or command string.

Entering the **alias** command without parameters displays all current aliases and their definitions. Aliases predefined by Pdbx can also be redefined. Refer to Chapter 3 for more information on predefined aliases.

You can nest aliases. In the following example, both **st** and **stp** are recognized as aliases by Pdbx.

```
(pdbx) alias st status (Return)
(pdbx) alias stp "st;ps"(Return)
(pdbx) stop at 14 (Return)
[1] stop at 14
(pdbx) run 3 (Return)
  %2  Stopped at breakpoint 1 in main at line 14
   14      if (n<1 || n>12) {
(pdbx) stp (Return)
[1] stop at 14
* %2  Stopped at breakpoint 1 in main at line 14
(pdbx) cont;st (Return)
3! = 1
  %2  Stopped by Exit in _exit at 0x314b
0000314b  ret
[1] stop at 14
```

The **unalias** command removes the definition of the specified alias. You can redefine an alias without removing its current definition.

Specifying Parameters with Aliases. *cmdstring* can reference parameters named in the alias definition. For example, assume the alias **sc** was defined with the following command:

```
alias sc(x,y) "stop in x if y"
```

At this point, entering the command **sc(sub1, x<1)** expands to **stop in sub1 if x<1**

When defining and invoking aliases that contain parameters, spaces may be used freely to separate words and punctuation.

LIMITATIONS

An alias name is not recognized if it is not the first word of a command.

```
assign variable = expression
```

variable — The name of the variable that receives the value defined in *expression*. *variable* can be an array element, structure field, register, or any other object to which a value can be assigned.

expression — An expression whose value is compatible with the data type of the specified variable.

DESCRIPTION

The **assign** command assigns the specified value to the specified program variable or register. Refer to Section 5.3 for an explanation of the syntax of variables and expressions.

EXAMPLES

Entering the following command stores the value 0xff in register `eax`:

```
assign $eax=0xff
```

Entering the following command assigns the value -1 to the variable *n* in procedure **main()**:

```
assign main.n = -1
```

The following command assigns the value 0.0 to element 1 of array **a**:

```
assign a[1] = 0.0
```

You can relax the strict type-checking done by the Pdbx **assign** command by setting the *$nostrict* debugger variable with the following command:

set $nostrict

For more information on predefined debugger variables, refer to the **set** command description.

LIMITATIONS

If you are debugging a C function and try to assign an enumerated constant to an enumerated variable, Pdbx appears to make the assignment, but does not.

You cannot conveniently assign constant strings to C character pointers or arrays. For example, the following command assigns *cp* the value "abc" instead of assigning *cp* to point to "abc".

```
assign cp = "abc"
```

Also, if *carr* is declared as `char carr[10]`, the following **assign** command causes Pdbx to complain that the variable and the expression are of incompatible types:

```
assign carr = "abc"
```

You can use the following procedure to assign constant strings to C character pointers or arrays.

1. Display the address where the string is stored.
2. Display the string in that memory location, both in character format and in words.
3. Patch in the new string, making sure that the string is not longer than the one it is replacing, and any characters that are not overwritten with the new string are deleted.

EXAMPLES

In this example, the pointer **cp** is used to replace the string "hello world" with the string "test". The program is stopped at a breakpoint and *$hexints* has been set. First, the address (memory location) where the string is stored is displayed in hexadecimal format.

```
(pdbx) print &*cp Return
0x1170
```

Next, the string in that memory location is displayed, both in character format and in words.

```
(pdbx) 0x1170/11c Return
'h' 'e' 'l' 'l' 'o' ' ' 'w' 'o' 'r' 'l' 'd'
(pdbx) 0x1170/3x Return
00001170:  6568 6c6c 206f
```

Then, the new string is patched as a 32-bit integer, making sure that the new string is no longer than the string it is replacing; any characters that would not be overwritten with the new string are overwritten with zeros.

```
(pdbx) assign 0x1170 = 0x74736574 Return
(pdbx) assign 0x1174 = 00000000000000 Return
```

Finally, the example uses the **print** command to display the new string.

```
(pdbx) print cp Return
"test"
```

```
call procedure ()
    or
call procedure ( parameters )
```

procedure The name of the procedure to execute.

parameters The set of parameters, separated by commas, to pass to the procedure. You must enter parentheses even if there are no parameters.

DESCRIPTION

The **call** command executes the specified procedure, passing it the specified parameters. In general, any procedure can be called from any point in the program. However, you should run the program and stop at a breakpoint *before* calling a subroutine. The debugging context is saved before the call and restored afterward, but any global or shared variables changed by the called procedure retain their new values.

If the called procedure is a function, you may want to use the Pdbx **print** command to execute the function and return its value—for example, **print factorial(5)**.

The procedure is executed in the current process, which must be stopped at the time of the **call** command. You cannot use the **call** command if the current process has reached the **exit()** system call. To return from a called procedure, use the **cont** command rather than **return**.

If the called procedure contains a breakpoint, execution of the **call** command stops at the breakpoint. (A **where** command at this point traces back only to the point of the **call** command.) While the called procedure is active, you can set and delete breakpoints, single-step the process, and so forth, in the usual fashion. When the procedure returns, Pdbx displays the following message:

```
procedure procname returned normally.
```

You can relax the strict type-checking done by the Pdbx **call** command by setting the *$nostrict* debugger variable with the following command:

```
set $nostrict
```

For more information on predefined debugger variables, refer to the **set** command description in this chapter.

NOTE

The **call** command is currently implemented for C programs only.

```
catch [event]
      [signal]
```

event `fork`, `exec`, or `exit`. You can enter an event name in either lowercase or uppercase.

signal The name or number of any signal listed in the **signal**(2) man page. You can enter the name of a signal in either uppercase or lowercase, with or without the `SIG` prefix.

DESCRIPTION

The **catch** command directs Pdbx to stop the program and return control to the terminal when the specified signal or event occurs in the current process. When invoked without parameters, **catch** lists all events and signals that Pdbx is catching.

By default, Pdbx catches execs and exits and all signals except `SIGHUP`, `SIGCONT`, `SIGCHILD`, `SIGALRM`, and `SIGKILL`, with the following exceptions:

- When Pdbx is invoked as dbx, it does not catch execs and exits of child processes.
- If the program being debugged is linked to the Parallel Programming Library and contains variables that reside in shared memory, Pdbx automatically ignores `SIGSEGV` signals. Parallel programs use `SIGSEGV` signals to grow stack and data space in shared memory.

The set of signals and events to be caught by Pdbx is inherited across forks and execs.

Pdbx delivers the "caught" signal when program execution is resumed with the **cont** command. You can suppress delivery of the signal by using the command **cont 0**.

If you direct Pdbx to catch forks, the child process resulting from a **fork()** system call is stopped just before it begins execution. The parent process is not stopped.

If you direct Pdbx to catch execs, the process is stopped just before it begins executing the new program resulting from an **execve()** system call.

If you direct Pdbx to catch exits, the process is stopped just before it finishes the **exit()** system call. At that point, you can examine the state of the process, but cannot prevent the process from exiting. If the process's parent is waiting for the process's exit (with a **wait()** system call or a `SIGCHILD` signal handler), the parent is not notified until execution of the process is continued after the stop.

The effect of the **catch** command can be reversed with the **ignore** command.

```
cont [%procnum | all] [signal] [to sourceline] [&]
```

procnum — The number of the process that is to resume execution. Defaults to the current process.

`all` — Specifies that all processes are to resume execution. Must be lowercase.

signal — The name or number of any signal listed in the **signal**(2) man page. You can enter the name of a signal in either uppercase or lowercase, with or without the `SIG` prefix.

sourceline — An integer value that specifies the source line where execution will be halted. *sourceline* can be in the current source file or another source file. To specify a line in a different source file, use the construct *"filename" : linenumber*. Execution halts each time the source line is encountered and just *before* it is executed.

`&` — Specifies that the process is to run in the background. Pdbx prompts for the next command without waiting for the process to stop.

DESCRIPTION

The **cont** command continues execution of the specified process in its current context.

If you specify *signal*, it is delivered to the process before the process resumes execution, and any other signal previously caught by Pdbx is canceled. Otherwise, the caught signal, if any, is delivered. You can suppress delivery of a caught signal by specifying a signal number of 0 (zero).

When multiple signals are sent to a process, Pdbx catches and delivers them one at a time, starting with the lowest-numbered signal. If a process has multiple signals pending, the *signal* parameter of the **cont** command overrides only the signal that was caught. For example, suppose that process %1 is stopped and you send it two signals: `SIGBUS` (signal 10) and `SIGFPE` (signal 8).

```
(pdbx) signal %1 bus Return
(pdbx) signal %1 fpe Return
```

When you continue program execution, Pdbx catches the lower-numbered signal (SIGFPE) before execution resumes:

```
(pdbx) cont %1 Return
  %1  Stopped by Floating point exception (integer divide by zero)
      in . at 0x8a0
000008a0  sprd   sp,r6
```

You can cancel SIGFPE by specifying a signal of 0 in the **cont** command, but SIGBUS is not canceled:

```
(pdbx) cont %1 0 Return
  %1  Stopped by Bus error in . at 0x8a0
000008a0  sprd   sp,r6
```

Entering **cont %1 0** resumes execution and cancels SIGBUS, or **cont %1** allows delivery of SIGBUS.

LIMITATIONS

If a breakpoint occurs on a machine instruction that causes a trap, the **cont** command may not work as expected. For example, suppose you use the **stop** command to set a breakpoint, and the instruction at the breakpoint address causes a floating-point trap. If you use the **cont** command after the breakpoint is encountered, execution proceeds through the system trap code and exits. Issuing **cont 0** repeatedly causes Pdbx to catch the trap.

If you supply your own trap handler and set the signal that causes Pdbx to ignore the trap, entering the **cont** command should result in entry to the trap handler. However, if you set the breakpoint on the machine instruction (address) that causes the trap, entry to the trap handler does not occur. This is usually not a problem since code such as a divide-by-zero normally causes a trap, and placing a breakpoint with the **stopi at *line_number*** command normally does not place a breakpoint at the divide instruction.

```
create [execfile] [arguments] [ < infile] [ > outfile]
```

execfile	The name of the file that contains the program to execute. You must specify *execfile* if more than one executable file was specified on the **pdbx** command line with the **–O** option; otherwise, it must be omitted.
arguments	Command-line arguments to pass to the program.
infile	The name of the file to use as standard input. Defaults to Pdbx's standard input device (0).
outfile	The name of the file to use as standard output. Defaults to Pdbx's standard output device (1).

DESCRIPTION

The **create** command creates a process from the specified *execfile,* or the *execfile* specified when Pdbx was invoked, if there was only one.

The **create** command is like the **run** command, with the following exceptions:

- **create** does not terminate any existing processes
- **create** stops the new process as soon as it execs the specified program. To execute the new process, use the **cont** command.

```
delete cmdnumber ...
```

cmdnumber An integer value that specifies the breakpoint or tracepoint to cancel. Pdbx assigns *cmdnumber* when you set a breakpoint or tracepoint using the **stop** or **trace** commands.

DESCRIPTION

The **delete** command cancels the breakpoint or tracepoint associated with *cmdnumber*. To list all existing breakpoints and tracepoints, use the **status** command.

EXAMPLES

The following display shows the deletion of two of five breakpoints in the current program. Once the breakpoints are deleted, the current breakpoints and tracepoints are redisplayed using the **status** command.

```
(pdbx) status Return
[1] stop at 14
[2] stop at 21
[3] stop at 23
(pdbx) delete 1 2 Return
(pdbx) status Return
[3] stop at 23
```

```
down [ nlevels ]
```

nlevels An integer that specifies the number of levels down the stack to move. Defaults to **1**.

DESCRIPTION

The **down** command moves the current procedure designation to the procedure, function, or subroutine that is *nlevels* levels down the stack from the current procedure. The current procedure designation determines the context for interpreting identifiers. The **down** command moves you toward the currently active procedure, while the **up** command moves you toward the main program. The context of the new current procedure (as reported by the **where** command) is listed. If moving *nlevels* levels would move you past the currently active procedure, Pdbx does not change the current procedure designation.

Unlike the **func** command, the **up** and **down** commands do not change the current source file designation.

The **up**, **down**, and **func** commands do not affect the state of the process being debugged.

EXAMPLES

In the following example, the current process is stopped with several instances of a recursive procedure active. The current procedure is the one most recently called; references to `n` apply to the variable *n* in that procedure:

```
(pdbx) where Return
factorial(n = 2), line 17 in "fac.c"
factorial(n = 3), line 21 in "fac.c"
factorial(n = 4), line 21 in "fac.c"
factorial(n = 5), line 21 in "fac.c"
main(argc = 2, argv = 0xfff820, 0xfff82c), line 12 in "fac.c"
```

```
(pdbx) print n; func Return
2
factorial
```

Entering the command **up 4** takes you up four levels to the main program, where "n" means variable *n* in the main program:

```
(pdbx) up 4 Return
main(argc = 2, argv = 0xfff820, 0xfff82c), line 12 in "fac.c"
(pdbx) print n; func Return
5
main
```

The command **down 2** takes you down to the second instance of the recursive procedure:

```
(pdbx) down 2 Return
factorial(n = 4), line 21 in "fac.c"
(pdbx) print n; func Return
4
factorial
```

```
dump [ procedure ] [ > filename ]
```

procedure The name of the procedure, function, or subroutine for which variables are listed. Defaults to the current procedure. If the procedure name is ".", Pdbx lists all variables in all active procedures.

filename The name of the file that receives the output from the **dump** command. Defaults to standard output (1).

DESCRIPTION

The **dump** command prints the names and current values of all variables in the specified procedure.

Refer to Table 5-4 for more information about the object types Pdbx uses when displaying memory. Refer to Table 5-5 for information on Pdbx predefined variables.

```
edit [ sourcefile ]
    or
edit procedure
```

sourcefile The name of the source file to edit. Defaults to the current source file.

procedure The name of the procedure, function, or subroutine in the source file to edit.

DESCRIPTION

The **edit** command invokes the text editor specified by the environment variable **EDITOR** for the specified file. If you specify *procedure,* and the editor is capable, Pdbx places the cursor at the beginning of the specified procedure. The current file designation is not affected.

NOTE

Note that Pdbx creates a view of the program source code (that is, the mapping of machine instructions to source line numbers) when you compile the program. That view does not automatically change when you edit one or more source files. Therefore, editing a source file may make the source code inconsistent with Pdbx's tables of compile-time debug information. To restore consistency between the source code and the debugger, you should recompile and relink the program, and then invoke Pdbx.

```
file [[ %execfile ] sourcefile ]
```

sourcefile The name or pathname of the file to use as the current source file.

execfile Directs Pdbx to use the source file directory list for the program in *execfile* first when searching for *sourcefile*. If you omit this parameter, Pdbx uses the directory list for the current program first, then the other program's directory lists. Omit this parameter unless you specified multiple **–O** options when you invoked Pdbx. Refer to the discussion of the **use** command for more details.

DESCRIPTION

The **file** command directs Pdbx to treat *sourcefile* as the current source file. By default, Pdbx uses the current source file to interpret references to line numbers in Pdbx commands and to list source code with the **list** command. Also, Pdbx sets traces and stops in the program associated with the current source file.

NOTE

Pdbx accesses the exact name you specified on the compile command line as the value of sourcefile. *Thus,* sourcefile *may be a filename or a pathname (relative or absolute).*

Entering the **file** command without parameters causes Pdbx to print the name of the current source file.

The **func** command also changes the current source file designation.

```
func [ procedure ]
```

procedure The name of the procedure, function, or subroutine to use as the current procedure. If you omit this parameter, Pdbx prints the name of the current procedure.

DESCRIPTION

The **func** command directs Pdbx to treat the specified procedure as the current procedure and to treat the source file that contains that procedure as the current source file. Pdbx interprets identifiers in the scope of the current procedure. The current source file is the file Pdbx uses by default when interpreting references to line numbers in Pdbx commands and when listing source code with the **list** command.

Entering the **func** command does not affect the state of the process being debugged.

```
help
```

DESCRIPTION

The **help** command prints a one-line summary of frequently used Pdbx commands.

```
ignore [event]
       [signal]
```

event `fork`, `exec`, or `exit`. You can specify an event name in either uppercase or lowercase.

signal The name or number of any signal listed in the **signal**(2) man page. You can specify the name of a signal in either uppercase or lowercase, with or without the `SIG` prefix.

DESCRIPTION

The **ignore** command directs Pdbx not to stop program execution when the specified signal or event occurs in the current process. If a signal that Pdbx is ignoring is sent to a process, Pdbx delivers it without prompting for commands.

Entering **ignore** without parameters causes Pdbx to list all events and signals that Pdbx is ignoring. By default, Pdbx ignores forks and the signals `SIGHUP`, `SIGCONT`, `SIGCHILD`, `SIGALRM`, and `SIGKILL`, with the following exceptions:

- If Pdbx is invoked as dbx, it ignores all execs and exits of child processes.
- If the program being debugged is linked to the Parallel Programming Library and calls the **init_shbrk** routine, Pdbx automatically ignores `SIGSEGV` signals. Parallel programs use `SIGSEGV` signals to grow stack and data space in shared memory.

You can reverse the effect of the **ignore** command with the **catch** command. Refer to Chapter 3 for more information about signals.

```
{ list  }
{ listi } [ linenumber [ , linenumber ] ]
   or
{ list  }
{ listi } procedure
```

linenumber An integer value that specifies from which line the listing begins. *linenumber* must be a line in the current source file.

procedure The name of the procedure, function, or subroutine to list.

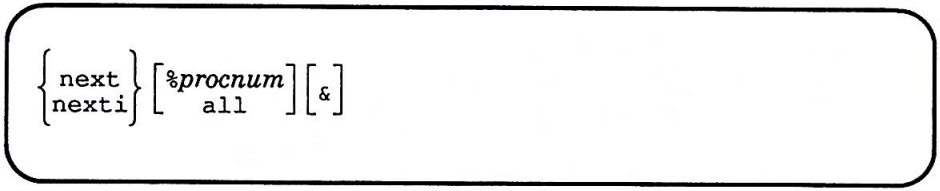
DESCRIPTION

The **list** command displays 10 lines at a time (or *$listwindow* lines, if set) starting from *linenumber* (or range of lines, if two numbers are given) in the current source file. The **listi** command is the same as the **list** command, except that it disassembles and displays the instructions that correspond to the specified source lines.

If you specify a procedure, Pdbx changes the current source file designation to the source file that contains the specified procedure, and lists *$listwindow* lines of code, centered around the first statement of the procedure. You can modify the *$listwindow* debugger variable using the Pdbx **set** command; *$listwindow* defaults to 10 lines and is used only when you specify *procedure*.

If you enter **list** without parameters, Pdbx displays ten or *$listwindow* more lines in the file, starting at the line following the last line listed by a **list** command, by a search command (*/pattern/* or *?pattern?*), or as part of a stop message. By default, **list** starts at the beginning of the current source file. Thus, you can repeatedly use **list** without parameters to read through a file.

If there is no current source file, use the **file** command to specify the file you want to list.



procnum	The number of the process (as displayed by the **ps** command) containing the next line or instruction to execute.
`all`	Specifies that the next line or instruction in all processes will be executed. Must be lowercase.
`&`	Specifies that the process is to run in the background: Pdbx prompts for the next command without waiting for the process to stop.

DESCRIPTION

The **next** command executes the next line of source code in the specified process. The default is the current process. Any caught signal is canceled, as with **cont 0**.

If the line to execute contains a call to a procedure (function or subroutine), Pdbx executes the procedure to completion. Otherwise, the **next** command has the same effect as the **step** command.

The **nexti** command is the same as the **next** command, except that **nexti** executes a single machine instruction.

```
print expression [ ,expression ] ...
```

expression Any valid Pdbx expression, as defined in Section 5.3. You can also specify the name of a compound variable (array, structure, or record) to print all components of the compound variable.

DESCRIPTION

The **print** command displays the values of the specified expressions. For example, entering the following prints the value of the variable *nextrow* in the current process and in process number 3:

```
print nextrow, 3:nextrow
```

Entering the following command prints the value of the stack pointer in the current process:

```
print $sp
```

Enter the following command to execute function **myfunc()** with the argument -1, and print the result:

```
print myfunc (-1)
```

The following command prints the contents of the variable of type "struct node" at address 0xfff000:

```
print *(0xfff000\&$$node)
```

Entering the following prints the value pointed to by *p*:

```
print *p
```

Pdbx expects pointer expressions to address bytes. If pointer *p* is declared as `int *p` and points to address 1000, you can use the following command to print the value at address 1001:

```
print *(ip+1)\&int
```

If you want to display the value at address 1004, you must compensate for the size of *p*. Since an object of type `int` is four bytes wide, use the following command:

```
print *(ip+4)\&int
```

The following command prints the fourth element of an array of doubles pointed to by "dp". Each element uses eight bytes of storage, so "4 * 8" is used to compute the location of the desired variable.

```
print *(dp + (4 * 8))\&double
```

If "dp" is already a pointer to double, the following command has the same effect:

```
print dp[4]
```

When Pdbx prints an expression, the expression is evaluated and placed on an internal stack. If the expression is a string, the string is copied on the internal stack. Long strings may cause Pdbx to display the message `expression too large to evaluate`, as in the following example:

```
char buf[100000];
main()
{
    strcpy(buf, "hello");
}
```

You would expect to be able to print *buf* after executing `strcpy`, but the declared size of *buf* is too large. Print the contents of the buffer with the command: **&buf/s**.

The **print** command displays the contents of a floating-point register as a floating-point number, unless the *$hexfloats* variable is set. You can set *$hexfloats* by by entering the following command:

```
set $hexfloats
```

Except when the 80387 is in use, double-precision numbers are stored in pairs of floating-point registers and single-precision numbers are stored in single floating-point registers. Since the debugger is unable to determine if a register contains a single-precision value or part of a double-precision value, double-precision is the default whenever possible. For example, if the first register of a register pair is **print**'s argument, the debugger assumes a double-precision value and displays the contents of the register pair as double-precision, even if the register pair contains single-precision values.

But if the second half of a double-precision register pair is **print**'s argument, single-precision is assumed and the register value is printed as single-precision.

You can use the following command to change the double-precision default to single-precision:

```
set $fpasingle
```

Invalid values may be reported for the floating-point register if any of the following statements are true:

- The second register in a double-precision register pair is given as the argument to **print**.
- An even floating-point register that contains a single-precision value is given as the argument to **print** and the *$fpasingle* variable is not set.

For the Symmetry Series, loading a Weitek register pair requires multiple instructions. When displaying a Weitek double-precision number between **stepi** instructions, it may occur that only half of the two-register pair has been loaded and the value displayed is invalid.

For information about setting other Pdbx predefined variables, refer to the **set** command description in this chapter.

EXAMPLES

```
(pdbx) list Return
    1   main()
    2   {
    3     char *ptr;
    4     ptr = (char *) 0x123245;
    5   }
(pdbx) stop at 5 Return
[1] stop at 5
(pdbx) run Return
  %2  Stopped at breakpoint 1 in main.main at line 5
    5   }
(pdbx) &ptr/X Return
3fffef20:  00123245
```

LIMITATIONS

Pdbx can display the value of a FORTRAN complex number, but supports no other `COMPLEX` operations.

Pdbx cannot print the name of a procedure to which a pointer points; however, you can use the shell's **nm** command to display the name of the procedure. For example, in the following C program fragment, line 2 declares a pointer to a function: `int (*f)();`.

```
 1
 2 int (*f)();
 3
 4 sub()
 5 {
 6 }
 7
 8 main()
 9 {
10     int i;
11
12     f = sub;
13     i= 1;
14     i++;
15 }
16
```

Now invoke Pdbx, set a breakpoint at line 14, and run the program. When the program stops at line 14, enter the **print** command to find the value stored in the pointer `f`. Then use the Pdbx **suspend** command to access the shell from which you invoked Pdbx. Search for that address in the binary file with the **nm** command.

```
(pdbx) stop at 14 Return
[1] stop at 14
(pdbx) run Return
  %2  Stopped at breakpoint 1 in main at line 14
   14     i++;
(pdbx) print f Return
0x114
(pdbx) suspend Return
Suspended
% nm -x a | grep 114 Return
sub                 |0x00000114|extern|          int( )|0x0008|      |.text
% fg Return
```

pdbx a

```
(pdbx) quit Return
```

```
print fpustack
```

DESCRIPTION

The **print fpustack** command displays control and data register information for the 80837 floating-point coprocessor. The Symmetry Series 80387 floating-point coprocessor contains eight floating-point registers, which are stored in a stack by the C and FORTRAN compilers. 80387 register values are not valid until the process executes. When you first invoke Pdbx on a process, you can print the registers and assign values to them, but the values are lost when execution begins. If you want to load values, use the following commands before loading the value:

stop in main
run

Refer to Intel's *80387 Numerics Processor Architecture Manual* for complete information on the 80387 floating-point registers.

The **print fpustack** command displays the following four control registers:

- Control word
- Status word
- Tag word
- Instruction pointer

The status word is decoded by the debugger to indicate the current top of stack (tos) and the tag word specifies the state of each floating-point register. The four floating-point register states are: valid, zero, invalid or infinity, and empty. If a floating-point register is empty when the **print fpustack** command is issued, only the control registers are displayed.

The instruction pointer is displayed with the label "ip=", and provides the address of the last instruction to be executed by the 80387 coprocessor.

The eight floating-point data registers are labeled from st(0) through st(7), where the st(0) register is always the current top of stack.

The display format for the floating-point registers includes the following values:

- Floating-point register label
- Hexadecimal representation of the floating-point number in the 80-bit format
- State of the floating-point register
- Decimal representation of the floating-point number

To display the entire set of floating-point registers, set the predefined variable *$fpustackall* with the following command:

```
set $fpustackall
```

If this variable is set, the **print fpustack** command may display floating-point data in the registers even though the state of the register is empty. This floating-point data is left over from previous calculations.

Since the C and FORTRAN compilers use the set of floating-point registers as a stack that is normally empty, issuing the **print fpustack** command between statements typically displays the stack as empty. If the stack of floating-point registers is displayed between **stepi** instructions within a floating-point expression, floating-point values are seen in the registers.

EXAMPLES

For this example, assume that the following C program exists:

```
main()
   {
      double a;
      double b;
      double c;
      double d;

      a = 1.0;
      b = 2.0;
      c = 3.0;
      d = (a / b) * (c / b);
}
```

The following session uses the **listi 11** command to disassemble and print the instructions that correspond to the floating-point expression in line 11 of the C source code. Next, the program is run and stopped at line 11, where floating-point manipulation begins.

```
(pdbx) listi 11 Return
   11      d = (a/b) * (c/b);
00000144   fldl    0xec(%ebp)
00000147   fld     %st(0)
00000149   fdivrl  0xf4(%ebp)
0000014c   fld     %st(1)
0000014e   fdivrl  0xe4(%ebp)
00000151   fmulp   %st,%st(1)
00000153   fstpl   0xdc(%ebp)
(pdbx) stop at 11 Return
[1] stop at 11
(pdbx) run Return
  %2  Stopped at breakpoint 1 in main at line 11
   11      d = (a/b) * (c/b);
```

Next, before executing line 11, the **print fpustack** command is entered to verify that the floating-point registers are empty. The assembly instructions within the floating-point expression are executed one at a time with the **stepi** command, and the floating-point registers are displayed after each **stepi**, using the **print fpustack** command. Note that the instruction displayed with the message `Stopped after step in main at line` *xx* is the next instruction to be executed, hence the contents of the registers before the execution of this instruction are displayed with the **print fpustack** command.

```
(pdbx) print fpustack Return
control word= 0x0232  status= 0x0000  tag= 0xffff  ip= 0x00000000 tos= 0
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
00000147  fld    %st(0)
(pdbx) print fpustack Return
 st(0)=  4000 8000 0000 0000 0000 [valid] = 2
control word= 0x0272  status= 0x3800  tag= 0x3fff  ip= 0x00000144 tos= 7
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
00000149  fdivrl 0xf4(%ebp)
```

```
(pdbx) print fpustack Return
 st(1)=  4000 8000 0000 0000 0000 [valid] = 2
 st(0)=  4000 8000 0000 0000 0000 [valid] = 2
control word= 0x0272  status= 0x3000  tag= 0x0fff  ip= 0x00000147 tos= 6
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
0000014c  fld    %st(1)
(pdbx) print fpustack Return
 st(1)=  4000 8000 0000 0000 0000 [valid] = 2
 st(0)=  3ffe 8000 0000 0000 0000 [valid] = 0.5
control word= 0x0272  status= 0x3000  tag= 0x0fff  ip= 0x00000149 tos= 6
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
0000014e  fdivrl 0xe4(%ebp)
(pdbx) print fpustack Return
 st(2)=  4000 8000 0000 0000 0000 [valid] = 2
 st(1)=  3ffe 8000 0000 0000 0000 [valid] = 0.5
 st(0)=  4000 8000 0000 0000 0000 [valid] = 2
control word= 0x0272  status= 0x2800  tag= 0x03ff  ip= 0x0000014c tos= 5
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
00000151  fmulp  %st,%st(1)
(pdbx) print fpustack Return
 st(2)=  4000 8000 0000 0000 0000 [valid] = 2
 st(1)=  3ffe 8000 0000 0000 0000 [valid] = 0.5
 st(0)=  3fff c000 0000 0000 0000 [valid] = 1.5
control word= 0x0272  status= 0x2800  tag= 0x03ff  ip= 0x0000014e tos= 5
(pdbx) stepi Return
  %2  Stopped after step in main at line 11
00000153  fstpl  0xdc(%ebp)
(pdbx) print fpustack Return
 st(1)=  4000 8000 0000 0000 0000 [valid] = 2
 st(0)=  3ffe c000 0000 0000 0000 [valid] = 0.75
control word= 0x0272  status= 0x3000  tag= 0x0fff  ip= 0x00000151 tos= 6
(pdbx) stepi Return
  %2  Stopped after step in main at line 13
00000156  fstp   %st(0)
(pdbx) print fpustack Return
 st(0)=  4000 8000 0000 0000 0000 [valid] = 2
control word= 0x0272  status= 0x3800  tag= 0x3fff  ip= 0x00000153 tos= 7
```

```
ps [ % procnum ]
```

procnum The process number for which Pdbx prints information.

DESCRIPTION

Entering the **ps** command without parameters causes Pdbx to print a numbered list of all processes, indicating the state of each. The current process is marked by an asterisk (*) in the left margin. After a process has called exit, the **ps** command issues the following message:

```
%n  Stopped by Exit in _exit at 0x1234
```

If you direct Pdbx to ignore exits, or allow the process to finish exiting, **ps** prints the following message:

```
%n  Exited
```

Processes that have been terminated (by means of **terminate**, **run**, or **rerun**) are not listed.

If you specify a process number, Pdbx prints the status of the indicated process, plus its process identification and parent process. If the process was created directly by Pdbx, the parent is displayed as "[pdbx]". If the process is in a core file, no extra information is displayed because the process doesn't really exist.

EXAMPLES

```
(pdbx) ps Return
* %2  Stopped at breakpoint 1 in main at line 11
  %3  Stopped at breakpoint 1 in main at line 11
  %4  Stopped by Exit in _exit at 0x1234
  %5  Exited

(pdbx) ps %2 Return
* %2  Stopped at breakpoint 1 in main at line 11
  %2  pid 17031, parent [pdbx].
```

```
(pdbx) ps %3 Return
  %3  Stopped at breakpoint 1 in main at line 11
  %3  pid 17032, parent %2.
```

```
quit
```

DESCRIPTION

The **quit** command terminates any active processes and exits Pdbx.

```
release % procnum
```

procnum The number of the process to release.

DESCRIPTION

The **release** command releases the specified process from Pdbx control and allows it to continue executing as though it had never been under Pdbx control. The process cannot be debugged, but you can continue debugging the process that called it.

You must enter the **catch fork** command before entering the **release** command so that execution stops after the process has forked, but before it execs. If your program forks other programs before forking the setuid program, use the **cont** command to continue execution. Once the setuid program has forked, enter the **release** command. Entering the **release** command after the process has forked but before the process has execed allows the setuid program to run as it normally would when not under Pdbx control.

EXAMPLES

The following example uses the **catch fork** and **release** commands to catch processes when they fork and release the specified process.

```
% pdbx program Return
(pdbx) catch fork Return
(pdbx) run 3 & Return
(pdbx)    %3  Stopped by Fork in fork at 0x844b
0000844b  jc       0x8454
  %4  Stopped by Fork in fork at 0x844b
0000844b  jc       0x8454
(pdbx) ps Return
  %4  Stopped by Fork in fork at 0x844b
  %3  Stopped by Fork in fork at 0x844b
* %2  Running cont
(pdbx) release %3 Return
```

```
(pdbx) ps Return
  %4  Stopped by Fork in fork at 0x844b
* %2  Running cont
(pdbx) release %4 Return
 3! = 6
(pdbx)    %2  Stopped by Exit in __exit at 0x2d3f
00002d3f  ret
(pdbx) cont & Return
(pdbx) ps Return
* %2  Exited
(pdbx) q Return
```

```
return [ procedure ]
```

procedure The name of the procedure to return to. If you do not specify *procedure*, the process executes until it returns from the current procedure.

DESCRIPTION

The **return** command continues execution of the current process until it returns to the specified procedure. The specified procedure must be active.

```
{ run   } [ execfile ] [ arguments ] [ <infile ] [ >outfile ] [ & ]
{ rerun }
```

execfile	The name of the file containing the program to execute. You must include this parameter if you specified more than one *execfile* on the Pdbx command line with the **–O** option; otherwise it must be omitted.
arguments	Command-line arguments to pass to the program. Separate arguments with spaces.
infile	The name of the file *execfile* uses as standard input. Defaults to Pdbx's standard input device (0).
outfile	The name of the file *execfile* uses as standard output. Defaults to Pdbx's standard output device (1).
&	Specifies that the process is to run in the background: Pdbx prompts for the next command without waiting for the process to stop.

DESCRIPTION

The **run** command terminates any existing processes, creates a new process from the specified *execfile* (or from the *execfile* created when Pdbx was invoked, if there was only one), and begins execution of the new process.

The **rerun** command is identical to the **run** command, except that if **rerun** is invoked without *arguments*, it uses the *arguments* specified in the most recent **run** or **create** command.

To create a new process without terminating existing processes, use the **create** command.

```
set [ name [ = expression ] ]
    or
unset name
```

name — The name of the debugger variable to activate, define, or delete. *name* can be any identifier that does not appear in the program being debugged.

expression — The value to assign to the debugger variable. *expression* can be any expression that evaluates to a numeric, Boolean, pointer, or string value, or the address of a procedure or variable.

DESCRIPTION

The **set** command creates the specified debugger variable if it does not already exist and assigns the specified expression to the variable. If *name* is a predefined debugger variable, the variable is enabled, allowing control of specific debugger actions; you cannot specify a value for *expression*. Note that the expression can include the debugger variable itself. For example, assume you have a linked list of nodes, each of following type:

```
struct node {
        int             data;       /* node contents */
        struct node     *next;      /* pointer to next node */
};
struct node     *listhead;          /* head of linked list */
```

You can use the following commands to display the list one node at a time, starting at the head of the list.

```
(pdbx) set ptr = listhead Return
(pdbx) print *ptr Return
[First node displayed.]
```

```
(pdbx) set ptr = ptr.next Return
(pdbx) print *ptr Return
[Second node displayed.]

(pdbx) set ptr = ptr.next Return
... and so on ...
```

Note that if you enter **set** at this point, Pdbx lists *ptr* as *listhead.next.next* .

When invoked without parameters, **set** displays the values of all existing debugger variables.

The **unset** command deletes or disables the specified debugger variable.

In general, you can use a debugger variable wherever an expression is allowed.

Note that **set** assigns the *text* of the expression, not the *value* of the expression, to the variable. The **set** command verifies that the expression is syntactically valid, but the expression is evaluated by each subsequent command in which the debugger variable appears.

EXAMPLES

In the following example, the debugger variable *overflow* is set to a Boolean expression and then used.

```
(pdbx) set overflow = njobs > maxjobs Return
(pdbx) set Return
overflow        njobs > maxjobs
(pdbx) stop in getwork if overflow Return
```

Table 5-5 lists debugger variables that have meaning to Pdbx. A type of "Boolean" indicates that Pdbx checks for the existence of the variable, but ignores its value. All of these variables are nonexistent (unset) by default.

Table 5-5
Pdbx-Defined Debugger Variables

Name	Type	Meaning When Set
$fpasingle	Boolean	Pdbx displays floating-point registers as single-precision numbers.
$fpustackall	Boolean	Pdbx displays nonzero floating-point stack entries even if in empty state (80387 only).
$hexchars	Boolean	Pdbx prints characters in hexadecimal.
$hexfloats	Boolean	Pdbx displays floating-point stack registers in hexadecimal.
$hexin	Boolean	Pdbx interprets integers in command input as hexadecimal.
$hexints	Boolean	Pdbx displays integers in hexadecimal.
$hexoffsets	Boolean	Pdbx displays offsets from registers in hexadecimal.
$hexstrings	Boolean	Pdbx displays character arrays in hexadecimal.
$listwindow	integer	Default number of lines to display with the **list** command. If not set, Pdbx prints 10 lines. When used with **list** ***proc*** for listing a procedure, Pdbx lists *$listwindow* lines around the procedure.
$noframe	Boolean	Pdbx does not follow the chain of call frames on the stack. This allows access to global variables and register contents in code that does not have conventional stack frames.
$nostrict	Boolean	Pdbx relaxes type-checking rules for **call** and **assign** commands.
$octin	Boolean	Pdbx interprets integers in command input as octal. If $hexin is set, $octin is ignored.

Table 5-5
Pdbx-Defined Debugger Variables (cont.)

Name	Type	Meaning When Set
$whichreg	Boolean	Pdbx includes the register name with the information displayed by the **whatis** command for the specified variable. Your program must declare the storage class for the variable as type `register`. Valid for C programs only.

For more information about predefined debugger variables, refer to the **print** command.

```
sh [ commandline ]
```

commandline The command line the shell executes.

DESCRIPTION

The **sh** command creates a new shell and passes it the specified command line for execution. If you do not specify *commandline*, Pdbx places you in the new shell, where you can enter one or more shell commands. After you exit the shell, Pdbx prompts for your next debugger command. The value of the **SHELL** environment variable determines which shell is used.

To exit temporarily to the shell from which you invoked Pdbx, use the **suspend** command.

NOTE

Do not use the **sh** *command to execute a program whose setuid bit is set. Refer to Section 3.5 for more information on referencing setuid programs.*

```
signal { %procnum } signal
       {  all     }
```

procnum	The process number to receive the specified signal.
`all`	Specifies that all active processes are to receive the specified signal. Must be lowercase.
signal	The name or number of any signal listed in the **signal**(2) man page. You can enter the name of a signal in either uppercase or lowercase, with or without the `SIG` prefix.

DESCRIPTION

The **signal** command sends the specified signal to the specified process(es). Pdbx sends the signal immediately using the **kill**() system call, but the result of the signal depends on whether the process is stopped or running; whether the signal is handled, ignored, or blocked in the process; and whether Pdbx is catching or ignoring the signal.

Process Running; Pdbx Catches Signal. In this case, the process stops and Pdbx announces the signal and prompts for a command. Pdbx delivers the signal when execution is continued with the **cont** command unless it is suppressed with **cont 0**.

Process Running; Pdbx Ignores Signal. The process stops (unseen by the user) and Pdbx immediately delivers the signal and resumes execution of the process.

Process Stopped; Pdbx Catches Signal. When execution of the process is next resumed, the process is immediately stopped and Pdbx announces the signal and prompts for a command. Pdbx delivers the signal when execution is continued with the **cont** command (unless it is suppressed with **cont 0**).

Process Stopped; Pdbx Ignores Signal. When execution of the process is next resumed, the process immediately stops (unseen by the user) and Pdbx delivers the signal and resumes execution of the process.

Refer to the descriptions of the **catch** and **ignore** commands for more information about signals.

```
source filename
```

filename The name of a file that contains Pdbx commands.

DESCRIPTION

The **source** command executes the Pdbx commands in the specified file as if they had been entered at the terminal. The specified file can contain aliases, commands to catch or ignore signals, and commands such as **stop at 20**. The **source** command is useful for giving the same set of instructions to Pdbx again without having to enter them each time you invoke Pdbx.

You can create a file for use with the **source** command in two ways:

- Create a new file with your text editor, entering the information you want to give to Pdbx.
- Redirect the output from the **status** command to a file by entering the command **status** > *filename* to save the breakpoint instructions you have given to Pdbx in a file so that you can use them again. Edit the file to add other instructions you may want to give to Pdbx, separating each command with a newline.

The following sample file was created by redirecting the output from a **status** command and editing the file to add some alias and ignore instructions. (You could use this file with the FORTRAN program *pfac.f* in Chapter 1.)

```
alias p "print argbuf,n,factorial"
ignore exit
ignore segv
stop at "pfac.f":7
stop at "pfac.f":8
stop at "pfac.f":13
stop at "pfac.f":20
```

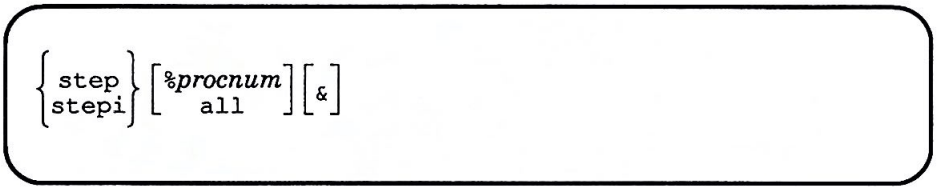
For information about invoking Pdbx using the *.dbxinit* initialization file, refer to Chapter 2.

```
status [ >filename ]
```

filename The name of the file to receive the output of the **status** command. Defaults to Pdbx standard output (1).

DESCRIPTION

The **status** command prints a numbered list of all currently defined breakpoints, tracepoints, and the command number associated with each. Use the **delete** command to cancel a breakpoint or tracepoint.



procnum	The number of the process containing the next line to execute.
`all`	Directs Pdbx to execute the next line in all processes. Must be lowercase.
`&`	Specifies that the process is to run in the background: Pdbx prompts for the next command without waiting for the process to stop.

DESCRIPTION

The **step** command executes the next line of source code in the specified process (default is the current process). Any caught signal is canceled, as with **cont 0**.

If the line to execute contains a call to a procedure (or function or subroutine), **step** stops at the beginning of the procedure. Otherwise, the **step** command has the same effect as the **next** command.

The **stepi** command is the same as the **step** command, except that **stepi** executes a single machine instruction.

```
stop [ procid ] in procedure [ if condition ]
   or
stop [ procid ] at sourceline [ if condition ]
   or
stop [ procid ] variable [ if condition ]
   or
stop [ procid ] if condition
   or
stop all
   or
stopi [ procid ] [ at ] address [ if condition ]
```

procid The process number where breakpoints are set. If you omit this parameter, Pdbx inserts the breakpoint into all processes associated with the current source file and stops all processes that encounter the breakpoint. This parameter can take three forms:

- **all**—stops all processes upon encountering the breakpoint in any process. Note that some processes may execute a significant amount of code before stopping. However, the process that encounters the breakpoint is stopped immediately.
- *%n*, where *n* is an integer value specifying the process number (as displayed by the **ps** command). The breakpoint is set only in process number *n*, and Pdbx stops only process *n*.
- *%filename*, where *filename* is the name of the executable file of a program being debugged. The breakpoint applies to all processes created from the program in the specified executable file. Use this form only when more than one program is being debugged in the same Pdbx session and

you want to set a breakpoint in a program other than the current one.

procedure The name of a procedure, function, or subroutine where the breakpoint is set.

sourceline A source line where the breakpoint is set. An integer value specifies a line in the current source file. To specify a line in a different source file, use the form *"filename":linenumber*.

variable The variable on which the breakpoint is set.

address The address where the breakpoint is set. Refer to Section 5.3 for a discussion of address syntax.

condition A simple Boolean expression, as defined in Section 5.3. (Boolean expressions with more than one operator give unpredictable results.) When a breakpoint is reached, the program is stopped only if the specified condition is true.

DESCRIPTION

The **stop** command directs Pdbx to stop program execution when entering the specified procedure, reaching the specified source line, or modifying or the specified variable. The **stopi** command directs Pdbx to stop program execution when reaching the instruction at the specified address or when the variable at the specified address is modified. The optional `if` *condition* clause directs Pdbx to stop program execution only if the specified condition is true.

As a special case, the command **stop all** immediately stops all running processes.

After interpreting a **stop** or **stopi** command, Pdbx displays its interpretation. For example, if *j* is a variable in procedure **main()** and you enter the command **stop j**, Pdbx displays the following message:

```
[1] stop j in main
```

The [1] is the command number associated with that breakpoint and can be used in a subsequent **delete** command to cancel the breakpoint.

After stopping the program, Pdbx displays the reason for the stop and the next line of source code it will execute—for example:

```
%1  Stopped at breakpoint 1 in myproc at line 13
 13             j = low;
```

Program execution can be resumed using a command such as **cont**, **step**, or **next**.

stop in procedure. This form of the **stop** command directs Pdbx to stop the program at the entry to the specified procedure. For example, entering the following command stops each process as it enters procedure **p_wait_barrier()**:

```
stop in p_wait_barrier
```

If an **if *condition*** clause is specified, the meaning of this command changes completely: when the procedure is called, Pdbx executes the procedure one line at a time, testing the specified condition after each line. If the condition is true, Pdbx stops the program. The program executes one line (then stop) after each subsequent **cont** command until the condition becomes false or the procedure exits. Program execution proceeds much more slowly when such a **stop** command is in effect.

stop at sourceline. This form of the **stop** command directs Pdbx to stop the program before executing the first statement on the specified source line. For example, entering the command **stop at 6** stops the program before executing the first statement on line 6 of the current source file.

stop variable. This form of the **stop** command directs Pdbx to stop the program and display the value of the specified variable whenever the variable's value changes. To detect all possible changes, Pdbx single-steps through the procedure that contains the variable, checking the variable's value after executing each line of source code. Program execution proceeds much more slowly when such a **stop** command is in effect.

stop if condition. This form of the **stop** command directs Pdbx to single-step through the program, testing the specified condition after each line of source code is executed. If the condition is true, the program is stopped.

stopi address. This form of the **stopi** command traces changes to the variable at the specified address. It is analogous to **stop *variable***.

stopi at address. This form of **stopi** directs Pdbx to stop the program before executing the machine instruction at the specified address. It is analogous to **stop at *sourceline***.

If a breakpoint occurs on a machine instruction that causes a trap, the **cont** command may not work as desired. For example, suppose the **stop** command is used to set breakpoint and the instruction at the breakpoint address causes a floating-point trap. If you use the **cont** command after the breakpoint is hit, execution proceeds through the system trap code and exits. Issuing **cont 0** repeatedly causes the trap to be caught, as expected.

If a user-supplied trap handler exists and the signal causing the trap is set to be ignored by Pdbx, the **cont** command should result in entry to the user-supplied trap handler. But if the breakpoint is set on the machine instruction (address) that causes the trap, entry to the trap handler does not occur. This is usually not a problem, since code such as a divide-by-zero normally causes a trap, and placing a breakpoint on the line of machine instruction with the **stopi at *address*** command normally does not place a breakpoint at the divide instruction.

Miscellaneous. If a process is running when a breakpoint is set, the breakpoint does not take effect in the running process until the process is stopped for some other reason.

If a statement consists of multiple lines, Pdbx traces only the last line of the statement.

As a by-product of the grammar of the **stop** command, the *procedure*, *condition*, and *variable* parameters can be specified using debugger variables, as in the following examples:

```
set aw = adjust_widget
set ovflow = next_widget > max_widgets
stop in aw if ovflow
```

EXAMPLES

In the following display, Pdbx stops the program before it executes the instruction at address 0x1160.

```
(pdbx) stopi at 0x1160 Return
[1] stopi at 4448
(pdbx) run 3 Return
  %2  Stopped at breakpoint 1 in factorial at line 23
   23   {
(pdbx) next Return
  %2  Stopped after next in factorial at line 24
   24      if (n = 1)
(pdbx) print $eax Return
0x3
```

```
suspend
```

DESCRIPTION

Entering the **suspend** command suspends execution of Pdbx and returns control to the shell from which Pdbx was invoked. You can use the shell's **fg** command to resume Pdbx operation.

NOTE

The **suspend** *command is not supported by the Bourne shell. Use the* **sh** *command to temporarily exit your debugging session.*

The Pdbx **suspend** command is analogous to the C shell's **suspend** command. Unlike most other programs, Pdbx cannot be suspended by means of the `SIGTSTP` signal (typically Ctrl-Z).

If a process is running when Pdbx is suspended, it continues to run until it hits a breakpoint or receives a signal.

terminate [*%procnum* | `all`]

procnum The number of the process to terminate.

`all` Directs Pdbx to terminate all processes.

DESCRIPTION

The **terminate** command terminates the specified process (defaults to the current process) and removes it from the process list. This command is useful is you are debugging a program linked to the Parallel Programming Library. Use **terminate** *%procnum* to end selected processes or use **terminate all** to terminate all processes.

```
trace [ procid ] [ in procedure ] [ if condition ]
   or
trace [ procid ] sourceline [ if condition ]
   or
trace [ procid ] procedure [ in procedure ] [ if condition ]
   or
trace [ procid ] expression at sourceline [ if condition ]
   or
trace [ procid ] variable [ in procedure ] [ if condition ]
   or
tracei [ procid ] [ in procedure ] [ if condition ]
   or
tracei [ procid ] address [ if condition ]
   or
tracei [ procid ] variable [ at address ] [ if condition ]
```

procid The process or program to trace. If you omit this parameter, the trace command applies to all processes, present and future, associated with the current source file. This parameter can take two forms:

- %*n*, where *n* is an integer value specifying the process number (as displayed by the **ps** command) to trace
- %*filename*, where *filename* is the name of the executable file of a program being debugged. The **trace** command applies to processes created from the specified program. Use this form only when more than one program is being debugged in the same Pdbx session and you want to set a breakpoint in a program other than the current one.

procedure A procedure, function, or subroutine. The clause "`in` *procedure*" specifies that tracing occurs only when the specified procedure is active. If you specify **trace** and **tracei** for the same procedure, Pdbx produces duplicate `entering function` messages.

sourceline The source line to trace. An integer value that specifies a line in the current source file. To specify a line in a different source file, use the form *"filename":linenumber*. The source line is printed each time it is encountered, just *before* it is executed.

variable The variable to trace. *variable* can be a simple variable, array, array element, structure/record, or structure/record field, and must be active when the **trace** command is entered. The variable's value is printed each time it changes.

expression Any valid Pdbx expression, as defined in Section 5.3. The expression is evaluated each time the specified source line (or address) is reached *before* the code at that point is executed and the value of the expression is printed.

address The address of an instruction to trace. Refer to Section 5.3 for a discussion of address syntax.

condition A simple Boolean expression, as defined in Section 5.3. (Boolean expressions with more than one operator give unpredictable results.) Any variables used with the **if** parameter must be active. When a breakpoint is reached, the program is stopped only if the specified condition is true.

DESCRIPTION

The **trace** and **tracei** commands direct Pdbx to print the specified trace information as the program is executed.

After interpreting a **trace** or **tracei** command, Pdbx displays its interpretation. For example, if *i* is a variable in procedure **main()** and you enter **trace i**, Pdbx responds with the following message:

```
[1] trace i in main
```

The [1] is the command number associated with that tracepoint and can be used in a subsequent **delete** command to cancel the tracepoint.

If a process is running when a trace option is set, the trace does not take effect in the running process until the process is stopped for some other reason.

The **tracei *address*** and **tracei *variable* at *address*** commands provide essentially the same trace information as the **trace *sourceline*** and **trace *variable* at *sourceline*** commands, respectively.

trace and **trace in procedure.** In this form of tracing, Pdbx prints each line of source code before executing it. All procedure entries and exits are also flagged as they occur. Program execution proceeds much more slowly when such a **trace** command is in effect.

If an **in *procedure*** clause is specified, tracing occurs only while the specified procedure is active; Pdbx sets breakpoints at the procedure's entry and exit points so that it knows when to turn tracing on and off. If no **in *procedure*** clause is specified, tracing is turned on for the duration of the procedure in which execution is currently stopped. If the program hasn't started yet, the entire program is traced. If the program is stopped in procedure *proc*, tracing is turned off when *proc* returns. The next time the same program is run, however, the entire program is traced.

tracei and **tracei in procedure.** These forms of the **tracei** command are similar to the corresponding **trace** commands, except that each machine instruction is disassembled and displayed before it is executed.

trace sourceline. This form of the **trace** command directs Pdbx to print the specified source line each time it is encountered and just before it is executed.

trace procedure. This form of the **trace** command directs Pdbx to trace calls to, and returns from, the specified procedure. Each time the procedure is called, Pdbx reports who called it and the values of the parameters, if any. When a traced function returns, Pdbx reports its value.

trace expression at sourceline. With this form of tracing, the specified expression is evaluated and printed each time the specified source line is reached and **before** the code at that line is executed. For example, entering the command **trace njobs < nworkers at 44** causes Pdbx to print the value of the expression *njobs < nworkers* before executing line 44 in the current source file.

trace variable. This form of the **trace** command directs Pdbx to print the value of the specified variable each time it changes. For example, entering the command **trace njobs** causes Pdbx to print the value of variable *njobs* each time its value is modified. To detect all possible changes, Pdbx steps through the program one line of code at a time, checking the variable's value after executing each line. Program execution proceeds much more slowly when such a **trace** command is in effect.

NOTE

If this command is in effect and program execution stops (due to a breakpoint or signal, for example), the **trace** *for the previous executed source line is not displayed. Work around this by checking the previously executed source line to see whether it modified the variable being traced.*

EXAMPLES

The following display shows the use of the **trace** command to print each line of code before it is executed in procedure **main()** and in all procedures called by **main()**. The integer following the colon is the source file line number.

```
(pdbx) trace in main Return
[1] trace  in main.main
(pdbx) run Return
%2  entering function main.main
%2  trace:      2   {
%2  trace:      4     ptr = (char *) 0x123245;
%2  trace:      5   }
%2  leaving function main.main
```

LIMITATIONS

If the **$INCLUDE** directive is used to include statements in a FORTRAN program (as opposed to declarations), Pdbx does not consider the included file to be part of the program source code. Tracing the included lines displays lines from the file containing the directive, rather than lines from the included file. You cannot set breakpoints in an included file.

When a statement consists of multiple lines, Pdbx traces only the last line of the statement.

```
up [ nlevels ]
```

nlevels An integer value that specifies the number of levels to move up the stack. Defaults to **1**.

DESCRIPTION

The **up** command moves the current procedure designation to the procedure, function, or subroutine that is *nlevels* levels up the stack from the current procedure. (The **down** command moves you toward the currently active procedure, while the **up** command moves you toward the main program.) Refer to the discussion of the **down** command for more information on the **up** and **down** commands.

```
use [ %execfile ] [ directory ... ]
```

execfile Specifies that this **use** command applies to the program whose executable file is *execfile*. Defaults to the program associated with the current process. Omit this parameter unless you specified multiple **-O** options when you invoked Pdbx.

directory The pathname of a directory where source files may be found. Constructs such as "~" and "~joe", which can be used in the C shell to designate users' home directories, are *not* recognized by Pdbx.

DESCRIPTION

NOTE

The **use** *command is necessary only when Pdbx cannot locate the source file for the current program. This is usually the case when a program is debugged in a directory other than the one where it was compiled* and *the source filename on the compile command line was relative to the current location. Refer to Chapter 3 for more information on using the* **use** *command.*

The **use** command specifies the list of directories to search for source files referenced by Pdbx commands. The specified list completely replaces the previous list. If no directory list is specified, **use** displays the directory list for the indicated program.

By default, the directory list contains the following directories, which are searched in the order listed:

- The directory in which the executable file resides
- Any directories specified using the **-I** option of the **pdbx** command
- The current directory (.)

```
whatis identifier
```

identifier The name of a procedure, function, subroutine, common block, variable, module, or similar program element.

DESCRIPTION

The **whatis** command displays the declaration of the specified procedure, common block, variable, or other program element.

You can display the register used to hold the *identifier* with the **whatis** command by setting the `$whichreg` debugger variable with the following command:

```
set $whichreg
```

For more information on predefined debugger variables, refer to the **set** command description in this chapter.

EXAMPLES

The following example shows that variable *m* in procedure **main()** is an `int`, and that **factorial()** is a function that takes an `int` parameter and returns a value of type `int`.

```
(pdbx) whatis main.m Return
int m;
(pdbx) whatis factorial Return
int factorial(n)
int n;
```

LIMITATIONS

The **whatis** command can display fields of C structures, but cannot display fields of arrays of structures (vs. a single structure).

```
where [ %procnum ]
```

procnum The number of the process for which a traceback is provided. Defaults to the current process.

DESCRIPTION

For each active procedure in the indicated process, the **where** command prints the name of the procedure, the values of the parameters passed to it, and the source line the procedure is stopped at. The most recently called procedure is listed first, and the main program is listed last.

If a **call** command is in progress, the **where** command traces back only to the point of the **call** command.

NOTE

An incorrect traceback may occur if you stop in a system call procedure. If the frame pointer is not pushed onto the stack, Pdbx issues the message `frame pointer not saved`. *You can use the* **where** *command to determine where you are located.*

EXAMPLE

For example, the following traceback indicates that **main()** called the recursive procedure **factorial()** with a value of 5, and **factorial()** has called itself three more times.

```
(pdbx) where Return
factorial(n = 2), line 13 in "fac.c"
factorial(n = 3), line 17 in "fac.c"
factorial(n = 4), line 17 in "fac.c"
factorial(n = 5), line 17 in "fac.c"
main(0x2, 0xfff820, 0xfff82c), line 7 in "fac.c"
```

```
whereis identifier
```

identifier The name of a procedure, function, subroutine, common block, variable, module, or similar program element.

DESCRIPTION

The **whereis** command lists the fully-qualified version of each program element with the specified name in the current program. Data types with the specified name are not listed. The order of the list is not meaningful.

NOTE

Because of limitations in the object file format, Pdbx cannot qualify global variables with the source file in which they are defined.

EXAMPLES

The following display shows that *n* is declared in procedures **factorial()** and **main()**, both of which are defined in source file *fac.c* .

```
(pdbx) whereis n Return
fac.factorial.n fac.main.n
```

```
which identifier
```

identifier The name of a procedure, function, subroutine, common block, variable, module, or similar program element.

DESCRIPTION

The **which** command reports which of several possible program elements with the specified name is referred to in the current context.

EXAMPLES

The following display shows the use of the **whereis** and **which** commands to identify Pdbx's interpretation of identifier *n* in the current context. Note that the **whereis** command identifies the contexts where *n* exists, and the **which** command identifies Pdbx's current interpretation of *n*.

```
(pdbx) whereis n Return
fac.factorial.n fac.main.n
(pdbx) which n Return
fac.main.n
```

Refer to Chapter 3 for information on providing the context Pdbx requires to recognize an identifier.

Appendix A
Error and Warning Messages

This appendix is an alphabetical listing of common warning and error messages produced by Pdbx. For each warning or error, the message text, an explanation of the message, and a recommended action are listed. This appendix does not include error messages that begin with the words "internal error". In this case, no corrective action exists, and you should file a mailbug against Pdbx.

–r flag is not implemented

Pdbx does not support the **–r** option. You cannot conditionally invoke the debugger.

***filename*: file already exists**

When attempting to redirect standard output, you specified a file that already exists.

Name the file something else. Note that >! to overwrite doesn't work.

***filename*: no corresponding object file '*objname*'**

The executable file corresponding to the core file specified by *filename* does not exist.

If you want to debug the core file, you must recompile the source for that file.

***filename*: not a core file**

The identifier specified after the executable filename is not a core file.

Verify that you have correctly typed the name of the file.

---

filename is not an object file

The specified file is not an executable file, or the file has been corrupted.

Reenter the command line, specifying the correct filename for the executable file that contains the program. If you correctly entered the name of the executable file, recompile the program, and then reinvoke the debugger.

---

identifier is not aliased

identifier is not the name of a currently defined alias.

Verify that you spelled the identifier correctly. If so, enter the **alias** command without parameters to display a list of currently defined aliases.

---

identifier is not set

The debugger variable you specified as an argument to the **unset** command is not currently set.

Use the **set** command to display the current settings for Pdbx debugger variables.

"_identifier_" is a program symbol -- use assign

The **set** command was used to try to assign a value to a program variable.

You must use the **assign** command to assign values to program variables. The **set** command assigns values to Pdbx predefined debugger variables only.

"_symbol_" is not a procedure or function

The specified *symbol* does not exist in any currently active procedure.

Use the **which** command to determine what *symbol* means in the current context. Use the **whereis** command to list all program elements with the name "symbol".

"*symbol*" is not active

symbol does not exist in any currently active procedure. Once entered, a procedure is active until it returns to the procedure from which it was called.

Verify that you have correctly typed the symbol name. If so, set a breakpoint in the procedure that contains *symbol* before attempting to print or examine its value.

"*symbol*" is not an array

Verify that you have correctly typed the name of the array. If so, use the **file** or **use** command (as appropriate) to correctly identify the source file that contains *symbol* or precede *symbol* with a process identifier.

"*symbol*" is not currently defined

The current context (for *symbol*) is not active.

Verify that you have typed the symbol name correctly. If so, set a breakpoint in the procedure that contains *symbol* before attempting to print its value.

"*symbol*" is not defined [(in program *progname*)]

The specified symbol does not exist in the current program. This message is printed when Pdbx attempts to disambiguate a symbol name.

Verify that you have typed the symbol name correctly. If so, use the **file** or **use** command (as appropriate) to correctly identify the source file containing **symbol** or precede *symbol* with the process identification.

"*symbol*" is not defined in "*block*"

symbol does not exist in "*block*", where *block* is a procedure, subroutine, or function. This message is printed when Pdbx attempts to disambiguate a symbol name.

Verify that you have typed the symbol name correctly. If so, use the **file** or **use** command (as appropriate) to correctly identify the source file containing **symbol** or precede *symbol* with the process identification.

"*symbol*" was not compiled with '–g'

The program unit represented by *symbol* was not compiled with the **–g** option. Consequently, only machine-level debugging is possible.

Can't continue program image (*process_id*)

The **continue** command was entered, but no program has been run yet. The only existing process, the program image, cannot be continued.

Use the **run** command to run the appropriate program.

Coff section count is zero.

The executable file has been corrupted.

Recompile your program, then reinvoke the debugger.

No process with id *process_id*

The process represented by *process_id* doesn't exist.

Use the **ps** command to list the current processes.

Process is not stopped

The process has exited or is still running.

Use the **ps** command to list all processes. If you need to create a process, use the **run** or **create** command. If you need to stop a running process, use the **stop all** or **signal** command.

---

alias expansion too large

While expanding alias in the command line, the number of characters in the expanded command line exceeded the limit of 10240.

Use multiple commands to accomplish the same task.

---

bad print format "*format*"

An incorrect format was used when displaying memory.

Refer to the table of formats in Chapter 5 or the *Pdbx Quick Reference Card*.

---

can't trace expressions

An expression was specified as an argument to the **trace** command. This is not supported.

Refer to the description of the **trace** command in Chapter 5 for more information.

---

core: No such file or directory

A file called *core* does not exist in the current directory.

Verify the location of the desired core file, move to the appropriate location, and reenter the command.

---

could not malloc coff section header space

An internal call to **malloc** failed due to insufficient memory.

Repeat the debugging session when sufficient memory exists, or use the **-Q** option on the Pdbx command line to reduce the amount of memory required by the program.

---

first address larger than second

The first address in a range of addresses specified with the / command is less than the second address—thus, no range exists.

Specify a valid address range.

---

havefile() called with nil pointer

Pdbx has lost track of which file is the current source file.

Use the **file** command to identify the current source file.

---

incompatible types

The value you attempted to assign to the specified variable or constant was declared as a different data type.

Specify a value of the correct type or use the **set $nostrict** command to relax type-checking rules.

---

missing command file name for –c (fatal)

The required argument to the **–c** option was missing on the command line.

Invoke Pdbx again, specifying the name of the command file to execute immediately after the **–c** option.

---

missing core file name for –C (fatal)

The required argument to the **–C** option was missing on the command line.

Invoke **pdbx** again, specifying the name of the core file immediately after the **–C** option.

---

missing directory for –I (fatal)

The required argument to the **–I** option was missing on the command line.

Invoke **pdbx** again, specifying a directory name immediately after **–I**.

---

missing object file name for *string* (fatal)

The required argument to the **–O** or **–Q** option was missing from the command line.

Invoke **pdbx** again, specifying the name of the object file on the command line, immediately after **–O** or **–Q**.

---

no executable code at line number *line_number*

The code corresponding to the specified line does not exist in the executable file. If the line number was entered correctly, this may be the result of optimization.

---

no source file

The current source file is not readable or there is no current source file. The latter can happen when the program being debugged has executed through its exit. At this point, Pdbx considers that no source file is active.

Use the **file** or **func** command to set the current source file or enter the **catch** command and rerun the program.

---

no source files were compiled with '-g'

No source files in the program unit being debugged were compiled with the **-g** option. Consequently, only machine-level debugging is possible.

---

non-terminated string

Missing a beginning or ending quote.

---

not enough parameters in macro call

The alias used in the command line was defined with more parameters than those specified.

Use the **alias** command to display all aliases and the definition for each.

---

not enough parameters to *symbol*

The number of parameters entered is less than the number of parameters defined for the specified procedure.

Verify the number and type of parameters to pass to the specified procedure and reenter the command line.

---

program "*prog_name*" has been modified -- symbolic information may be out of date

The source file containing the program specified by *prog_name* was modified.

Recompile the program and reinvoke the debugger.

---

symbol *symname* not callable

symname is not a procedure or Pdbx is confusing it with a source module that has the same name.

Use the **whatis** and **whereis** commands identify the fully qualified name of the procedure.

---

syntax error

The command is incorrectly spelled, the command line contains spurious characters, or the command references an identifier that is a Pdbx reserved word.

Reenter the command line. If the error message is displayed again, reenter the command line and enclose the identifier in back quotes.

---

unknown option '*char*' (fatal)

The character "–" preceded the invalid option "*char*".

Invoke **pdbx** again, specifying a valid command option. Refer to Chapter 5 for a list and description of all Pdbx command options.

Appendix B
Reserved Words

This appendix lists the identifiers that have special meaning in Pdbx depending on the context.

alias	edit	not	skip	up
all	file	on	source	use
and	fpustack	or	status	wait
assign	func	print	step	whatis
at	help	ps	stepi	when
call	if	psym	stop	where
catch	ignore	quit	stopi	whereis
cont	in	release	suspend	which
create	list	rerun	terminate	window
debug	listi	return	to	CHAR
delete	mod	run	trace	INT
div	next	set	tracei	NAME
down	nexti	sh	unalias	REAL
dump	nil	signal	unset	STRING

Appendix C
Differences from UNIX 4.3bsd dbx

C.1 Overview C-1
Unsupported Features C-1
Enhancements C-1
C.2 pdbx vs. dbx C-3
C.3 Command Line Syntax Changes C-4
C.4 Command Set Modifications C-4
C.5 Using Pdbx as dbx C-7

Figures

C-1 sys.c program listing C-8

Tables

C-1 Command Set Comparison C-4

Appendix C
Differences from UNIX 4.3bsd dbx

C.1 Overview

This appendix describes UNIX 4.3bsd **dbx** features not supported by Pdbx as well as the additional features Pdbx supports. This appendix also includes two sample debugging sessions that demonstrate these differences.

C.1.1 Unsupported Features

Pdbx does not support the following UNIX 4.3bsd **dbx** features:

- Suspension of the debugger with the `SIGTSTP` signal, typically Ctrl-Z. Pdbx ignores `SIGTSTP`, passing it to the running process, if any. Use the Pdbx **suspend** command to suspend the debugger.
- The **gripe** command. Report problems with Pdbx to Sequent using the **mailbug**(1) utility.
- The **–r** option. You cannot conditionally invoke Pdbx.
- The **–k** option. Pdbx does not support kernel debugging.

C.1.2 Enhancements

The major enhancements to Pdbx fall into the following categories. These topics are discussed in more detail throughout the remainder of this chapter.

- **multiple processes.** Pdbx can monitor multiple processes simultaneously. Typically, the processes are related, as a parent process and multiple child processes, but they may be completely unrelated. To aid you in the tracking of multiple processes, Pdbx assigns each process a number; the value 1 is assigned to the first process created. You can list process numbers with the Pdbx **ps** command and then specify which Pdbx commands apply to which processes.

You can start, stop, trace, and examine processes independently. You can set breakpoints in individual processes or in all processes. A breakpoint can stop all processes or only the processes that encountered the breakpoint.

You can direct Pdbx to stop program execution when a process forks (by means of the **fork()** system call), execs (by means of the **execve()** system call), or exits (by means of the **exit()** system call). These events may be caught or ignored like ordinary UNIX signals.

- **multiple executable files**. Different processes being debugged can run different programs or a process can exec (that is, begin running a new program). When a process execs, it inherits the context of the new program, including the current breakpoints, tracepoints, and source files. When invoking Pdbx to debug multiple executable files, you must specify the names of all programs to debug using the **-O** option or use the **-Q** option to specify which programs *not* to debug.
- **multiple core files**. You can examine multiple core dumps in the same debugging session. Use the **-C** option when you invoke Pdbx to specify multiple core files.
- **synchronous or asynchronous terminal operation**. UNIX 4.3bsd dbx handles terminal I/O in a synchronous fashion: while the process being debugged is running, all terminal I/O is associated with that process. dbx does not prompt for a command until the process is stopped by a breakpoint, Ctrl-C, or other event. When debugging parallel applications, it may be useful to enter commands while one or more processes are running. Therefore, Pdbx commands that put a process into execution (**run**, **rerun**, **cont**, **next**, **step**) accept an optional ampersand (**&**) parameter, which indicates that the process is to run in the background. After starting the process running, Pdbx immediately prompts for another command. If a process running in the background needs to read from the terminal, it stops executing and Pdbx displays the following message:

```
stopped (tty input)
```

Pdbx's handling of asynchronous terminal I/O is similar to that of the C shell.

C.2 pdbx vs. dbx

You can invoke Pdbx as either **pdbx** or **dbx**. Both commands are links to the same program, but invoking Pdbx as **dbx** causes the debugger to ignore all execs and to ignore exits of child processes.

Pdbx accepts the following command line options not recognized by UNIX 4.3bsd **dbx**:

–a	Directs Pdbx to run all processes asynchronously in the background. No process can read from the terminal.
–C *coredump*	Specifies *coredump* is a core file to analyze. Use when there is more than one program or more than one core file.
–O *execfile*	Identifies *execfile* as an executable program to debug. Use this option only when there is more than one program to debug.
–Q *execfile*	Identifies *execfile* as an executable program *not* to debug. The **–Q** option is similar to the **–O** option above, except that it specifies a program that contains minimal debugging information when loaded into Pdbx. (Refer to Chapter 3 for information on debugging with limited information.) This option is useful when debugging large programs and available memory space is limited.
–u	Preserves the case of identifiers for FORTRAN programs. The C compiler always preserves the case of identifiers. The FORTRAN compiler converts all identifiers to lowercase unless you specified the **–case** option on the compile command line. If you invoked the FORTRAN compiler with the **–case** option, you should invoke Pdbx with the **–u** option to preserve case sensitivity.

C.3 Command Line Syntax Changes

In Pdbx, you can prefix an expression with a process number to indicate which process is affected by the command. For example, entering the following command prints the value of *status_word* in process 3, regardless of which process is the current process.

```
print 3:status_word
```

You can also specify an ampersand (&) after some Pdbx commands to indicate that the process is to run in the background.

C.4 Command Set Modifications

The Pdbx command set includes commands that behave the same under both Pdbx and **dbx** and new and modified commands to support the enhancements listed in Section C.1.2. The following table lists all Pdbx commands, specifying which are new (not found in 4.3 bsd **dbx**), which behave the same as in 4.3bsd **dbx**, and which behave differently (and in what way).

Table C-1
Command Set Comparison

Command	New	Same	Additional Functionality
%n	✔		Change the current process to process number *n*.
alias		✔	
assign		✔	
call		✔	
catch			Catches events, such as forks, execs, and exits, as well as signals.
cont			Can specify which processes continue execution, an ampersand (&) to continue processes in the background, and the source line to which execution continues.

Table C-1
Command Set Comparison (cont.)

Command	New	Same	Additional Functionality
create	✔		Similar to **run**, but does not automatically execute the new process and does not terminate other processes.
delete		✔	
down		✔	
dump		✔	
edit		✔	
file			Can specify that the **file** command refers to a particular program when multiple programs are being debugged.
func		✔	
gripe			Not supported. Use the DYNIX/ptx **mailbug** command.
help			Lists frequently-used commands only.
ignore			Can ignore an event, such as a fork, exec, or exit, as well as signals.
list, listi		✔	
next, nexti			Can specify the process to step through. Can also specify an ampersand (**&**) to step process(es) in the background.
print		✔	
print fpustack	✔		Displays control and data register information for the 80837 floating-point accelerator.
ps	✔		Lists processes and the status of each. When debugging a core file, reports which signal aborted the program.

Table C-1
Command Set Comparison (cont.)

Command	New	Same	Additional Functionality
quit		✔	
release	✔		Releases a process stopped by a **catch fork** command. The released process executes without Pdbx control to allow succesful execution of setuid programs.
rerun			Can specify ampersand (&) to run the process in the background. Requires that you specify the name of the executable file if more than one program is being debugged.
return		✔	
run			Can specify ampersand (&) to run the process in the background. Requires that you specifiy the name of the executable file if more than one program is being debugged.
set			Pdbx does not support the *$frame*, *$mapaddrs*, *$unsafeall*, and *$unsafeassign* debugger variables.
sh			If you do not specify *command-line*, Pdbx executes a new shell.
signal	✔		Sends a signal to a process or to all processes.
source		✔	
status		✔	
step, stepi			Can specify the procecess(es) to step through. Can also specify an ampersand (&) to step process(es) in the background.

Table C-1
Command Set Comparison (cont.)

Command	New	Same	Additional Functionality
stop, stopi			Can specify the process to stop. Can stop all processes when any process encounters a breakpoint. The **stop all** option stops all running processes immediately.
suspend	✔		Suspends execution of Pdbx and returns control to the shell. The SIGTSTP signal (typically Ctrl-Z) does not suspend Pdbx.
terminate	✔		Terminates the current process, the specified process, or all processes.
trace, tracei			Can specify the process to trace.
unalias		✔	
unset		✔	
up		✔	
use			Can specify which program the **use** command applies to.
whatis		✔	
where			Can specify the process for which the traceback is printed.
whereis		✔	
which		✔	

C.5 Using Pdbx as dbx

This section presents two debugging sessions that illustrate how Pdbx's operation differs when it is invoked as **dbx**. Invoking Pdbx as **dbx** is especially useful when debugging a program that creates multiple processes and only the parent process is of interest. Invoking the debugger as **dbx** allows descendants of the program being debugged to execute without debugger intervention.

Figure C-1 lists the C program *sys.c*, which is used in the sample sessions.

```
 1 /* sys.c -- use the "system" routine */
 2
 3 #include <stdio.h>
 4
 5 main()
 6 {
 7     int i;
 8
 9     printf("about to call system()\n");
10     i = system("./xyz");
11     printf("system() returns %d\n", i);
12 }
```

Figure C-1. sys.c program listing. In this program, the **system** utility creates a child process, the shell *sh*, which receives the string "./xyz" as an argument. The program then prints the value returned from the call to **system()**.

The file *./xyz* contains the following two lines:

```
#! /bin/sh
echo "xyz running"
```

NOTE

Be sure the file permissions for ./xyz allow execution of the file.

Compile the program using the **cc** command, invoke Pdbx as **dbx**, and run the program.

```
% cc -g -o sys sys.c Return
% dbx sys Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(dbx) run Return

about to call system()
xyz running
system() returns 0
  %2  Stopped by Exit in _exit at 0x2eb
000002eb  ret
(dbx) quit Return
```

The program executes as expected, with **dbx** catching the exit of the parent process, but not exits of child processes.

When you invoke Pdbx as **pdbx**, processes are stopped when they begin execution, when they exit, and when they receive a signal that is not being ignored. To duplicate the output from the previous session when using **pdbx**, you must use the **ignore** command to instruct Pdbx to continue execution when these events occur. Note that the output in the following debugging session does not exactly duplicate the output from the previous session. Entering the **ignore exit** command causes Pdbx to ignore *all* exits, including the exit of the parent process.

```
% pdbx sys Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) ignore exec Return
(pdbx) ignore exit Return
(pdbx) run Return

about to call system()
xyz running
system() returns 0
(pdbx) quit Return
```

To track the creation and demise of processes that are generated when you run **sys**, invoke the debugger as **pdbx** and enter the command **run &** at the (pdbx) prompt. Including an ampersand (**&**) on the command line directs Pdbx to run the program in the background, which allows Pdbx to accept subsequent terminal input without passing it to the executing program.
Note that since the process runs in the background, it is running asynchronously with the foreground process, and output from foreground and background processes may be intermixed.

```
% pdbx sys Return
dbx version x.x of 8/22/89 18:12 (Sequent).
Type 'help' for help.
reading symbolic information ...
(pdbx) run & Return

about to call system()
  %3  (sh ) Stopped by Exec in . at 0x0d0
000000d0  subl   $0x8,%esp
(pdbx) ps Return
  %3  (sh)  Stopped by Exec in . at 0x0d0
* %2  (sys) Running cont
```

The output from the **ps** command shows that process 2 is running the **sys** program and process 3 is stopped immediately after the **execve** system call. (By default, the debugger ignores forks, but catches execs.) The **system** utility, which is referenced in the **sys** program, forked and exec'ed a shell. That shell will then fork and exec the shell script in the file *xyz*.

To continue execution of process 3, use the **cont** command as shown in the following display. Then enter the **ps** command to list the current status of each process.

```
(pdbx) cont %3 & Return
(pdbx) ps Return
  %4   (xyz) Stopped by Exec in . at 0x0d0
000000d0  subl   $0x8,%esp
  %4   (xyz) Stopped by Exec in . at 0x0d0
  %3   (sh ) Running cont
* %2   (sys) Running cont
```

The output from the **ps** command shows that process 4 has been stopped by an exec. Processes 3 and 2 are still running.

Use the **cont** command to resume execution of process 4.

```
(pdbx) cont %4 & Return
xyz running
  %4   (xyz ) Stopped by Exit in . at 0xc387
0000c387   ret
(pdbx) ps Return
  %4   (xyz ) Stopped by Exit in . at 0xc387
  %3   (sh  ) Running cont
* %2   (sys ) Running cont
```

When process 4 resumes execution, the shell script in file *xyz* executes, and the message `xyz running` is printed at *stdout*. The other two processes are running: process 3 is waiting for its child (process 4) to exit and process 2 is waiting for its child (process 2) to exit.

Use the **cont** command to direct Pdbx to resume execution of process 4, which is now stopped at the **exit** call.

```
(pdbx) cont %4 & Return
  %3  (sh ) Stopped by Exit in . at 0xc387
0000c387  ret
(pdbx) ps Return
  %4  (xyz) Exited
  %3  (sh ) Stopped by Exit in . at 0xc387
* %2  (sys) Running cont
```

At this point, process 4 has exited and now process 3 is stopped. Use the **cont** command to continue each process until all have exited. As each process exits, the parent process verifies that the child has exited. Once all child processes have exited, the parent finishes executing and exits. Note that an ampersand is not used with the final **cont** command. Since process 2 is the only process that has not exited, terminal input can be passed to the program.

```
(pdbx) cont %3 & Return
system() returns 0
  %2  (sys)  Stopped by Exit in _exit at 0x2eb
000002eb  ret
(pdbx) ps Return
  %4  (xyz) Exited
  %3  (sh)  Exited
* %2  (sys) Stopped by Exit in _exit at 0x2eb
(pdbx) cont %2 Return
(pdbx) ps Return
  %4  (xyz) Exited
  %3  (sh ) Exited
* %2  (sys) Exited
(pdbx) quit Return
```

Index

&, significance of 3-4
% command 5-16
/ command 5-12
  /pattern 5-12
  address/ 5-14
: (colon) operator 5-6
; (semicolon) in a command line 5-3
= command (*address*=) 5-14
? command 5-12
* (asterisk) in ps display 4-2
$$ operator 5-9
$fpasingle 5-42,58
$fpustackall 5-46,58
$hexchars 5-58
$hexfloats 5-41,58
$hexin 5-8,58
$hexints 5-58
$hexoffsets 5-58
$hexstrings 5-58
$listwindow 3-9; 5-38,58
$noframe 5-58
$nostrict 5-10,58
$octin 5-8,58
$whichreg 5-59
.dbxinit file 2-3
.o file 1-16; 2-2
_init_shbrk 1-17
–a option 4-4; 5-2
–c option 2-3
–C option 5-2
–c option 5-2
–g compiler option 1-5; 2-2
–I option 3-11; 5-2
–i option 5-2
–o compiler option 1-5
–O option 4-6; 5-2
–Q option 4-7; 5-3
–u option 5-3

SHELL environment variable 3-22
4.3bsd dbx unsupported features C-1
4.3bsd dbx vs. pdbx
  4.3bsd dbx unsupported features C-1
  command line option differences C-3
  command line syntax differences C-4
  command set differences C-4
  enhancements to pdbx C-1

A

Abbreviating commands 3-22; 5-4,17
Active procedures, listing 5-81
Active processes, terminating 3-8
Address of variable or procedure (& operator) 3-4
Address syntax 5-7
address/ and *address*= commands 5-14
Addressing physical address space 3-23
alias command 3-22; 5-4,17
Aliases, command
  creating 1-9; 5-4
  defining 5-17
  deleting 5-17
  displaying 5-17
  listing 1-9
  nesting 5-18
  ordering in display 1-9
  parameterized 5-18
Ampersand (&)
  address operator 3-4
  use of 4-4
Arguments of active procedures, displaying 5-81
Arithmetic expressions 5-7
Array indexing 3-4
Array syntax 3-4
assign command 5-19
  relaxing type checking for 5-19
Assigning a value to a debugger variable 5-56
Assigning a value to a variable 5-19
`assuming parallel library used` message 4-5
Asterisk (*) in ps display 4-2
Asynchronous and synchronous terminal I/O ..... 4-4 to 5

B

Back quotes, use with identifiers 3-2; 5-4
Background execution of process ..... 4-4 to 5
Backslash operator 5-9
Barrier (mutual exclusion mechanism) 4-5
Blanks in command lines 5-4
Boolean expressions 5-7
Breaking when signal or event occurs 5-24
Breakpoints
  canceling 5-29
  deleting 1-11; 3-8
  displaying 3-8
  implications for parallel applications 4-1
  listing currently defined breakpoints 5-64
  on machine instructions 5-27
  setting 1-7; 3-8

C

C character pointers
  assigning 5-20
C shell and Pdbx commands for background execution 4-4
C$DOACROSS directive 1-22
C$DOACROSS loops
  debugging 4-6
  naming of 4-6
call command 5-22
  relaxing type checking for 5-22
Canceling breakpoints or tracepoints 5-29
Case of Pdbx commands 5-4
Casts (data type conversion) 5-9
catch command 3-24; 5-24
Catching signals 3-24
Character strings 5-8
Child processes 5-37
Clarifying scope of identifier 5-83
Coercion of data types 5-9
Colon (:) operator 5-6
Command line options
  –a 5-2
  –C 5-2

-I 5-2
-O 5-2
-Q 5-3
-u 5-3
ordering of 5-2,3
Command number 3-8
Commands ..... Chapter 5
abbreviating 5-17
aliasing 5-17
command line syntax ..... 5-3 to 11
listed by function 5-1
on-line synopsis of 5-36
Pdbx startup command file 5-63
release a process 5-52
Constant strings
assigning 5-20
cont command 5-26
Context for interpreting identifiers 3-22
context.*identifier* syntax 5-5
continue command 1-22
Continuing program execution 1-8; 5-26
Core dumps, analyzing 3-18
core files 3-18; 5-1
Crashing the system with Pdbx 3-23
create command 1-9; 3-6; 5-28
Creating
a process 1-9; 3-6
aliases 1-9
command aliases 5-4
Ctrl-C 3-24; 4-5
Ctrl-Z 3-21,24; 4-5
Current procedure 3-22
definition of 4-3
setting 5-30,35
Current process 4-2
definition of 4-2
setting 5-16
using **ps** to identify 5-49
Current program 4-3
definition of 4-2

Current source file
definition of 4-3
setting 5-34,35
Current source line 4-3
pattern search in source file 5-12
Custom devices, precautions for accessing 3-1

D

Data type conversion 5-9
dbx compared to Pdbx ..... Appendix C
Debugger variables
$fpasingle 5-58
$fpustackall 5-46,58
$hexchars 5-58
$hexfloats 5-58
$hexin 5-58
$hexints 5-58
$hexoffsets 5-58
$hexstrings 5-58
$listwindow 5-38,58
$noframe 5-58
$nostrict 5-10,58
$octin 5-58
$whichreg 5-59
assigning values to 5-57
Debugging examples
C program ..... 1-6 to 13
parallel FORTRAN program ..... 1-17 to 23
Debugging parallel applications ..... Chapter 4
Declaration of identifier, printing 5-80
Default name of executable file 2-6
Defaults
beginning of pattern search 5-12
caught signals 5-24
current process 4-2
ignored signals in dbx 5-37
ignored signals in Pdbx 5-37
lines of code displayed 5-38
name of executable file 2-6
Defining aliases for Pdbx commands 5-17

Definitions
core dump 3-3
current procedure 3-22; 4-3
current process 4-2
current program 4-2,3
current source file 4-3
current source line 4-3
dumping core 3-3
heterogeneous application 4-1
homogeneous application 4-1
parallel application 4-1
signal 3-24
delete command 1-11; 3-8; 5-29
Deleting
a process 3-7
aliases for Pdbx commands 5-17
breakpoints 3-8
Demonstration run
parallel FORTRAN program ..... 1-13 to 23
simple C program ..... 1-3 to 13
Designating calling procedure as current procedure 5-78
Designating the current procedure 5-35
Differences between Pdbx and previous versions of dbx ..... Appendix C
Directories where source files reside 5-79
Disassembling machine instructions 5-14,38
in trace output 5-77
Displaying
80387 registers 5-45
active stop commands 5-64
active trace commands 5-64
aliases for Pdbx commands 5-17
breakpoints 3-8
current status 3-8
declaration for identifier 5-80
ignored signals 5-37
source files 3-9
value of a variable or expression 5-40
value of variables 3-13
down command 3-22; 4-3; 5-30
dump command 3-13; 5-32
Dumping core, defined 3-3
Duplicate names 3-16

DYNIX/ptx commands, executing from Pdbx 5-60

E

edit command 5-33
Editing source file 5-33
Error and warning messages 3-2; A-1
Error recovery ..... 3-2 to 3
Examples
$listwindow variable, using 3-10
alias command, using 1-9; 3-22
compiling a C program 1-5
compiling a FORTRAN program 1-16
continue command, using 1-8,22
continuing program execution 1-8
create command, using 1-9
creating a new process 1-9
creating aliases 1-9
creating breakpoints 1-11
delete command, using 1-11; 3-8
deleting breakpoints 1-11
disambiguating identifiers 3-17
displaying a linked list 5-56
displaying function value 5-40
displaying values 3-13
displaying variable values 1-8; 5-40
dump command, using 3-13
file command, using 3-11
ignore command, using 1-7
invoking Pdbx 1-6
list command, using 1-7
print command, using 1-8
ps command, using 1-8; 3-9
quit command, using 1-22
release command 5-52
rerun command, using 1-7
run command, using 1-6; 3-7
running a program 1-5
setting breakpoints 1-7
status command, using 1-11
stop command, using 1-7
terminate command, using 3-7

trace command, using 1-11
tracing program execution 1-11; 5-76
unalias command, using 3-22
use command, using 3-12
Examples, debugging
C program ..... 1-6 to 13
parallel FORTRAN program ..... 1-17 to 23
Executing
next line or instruction 5-39
Pdbx commands from a file 5-63
selected procedure 5-22
shell command 3-21
shell commands from Pdbx 5-60
single source line 5-65
Exiting
a procedure 5-54
Pdbx 3-8; 5-51
Exits, program
ignoring 1-6
stopping at 1-6
`expression too large to evaluate` message 5-41
Expressions 5-6,11
arithmetic 5-7
Boolean (logical) 5-7
displaying value of 5-40

F

fac.c 1-4
factorial function 1-3
file command 3-10; 5-34
Filenames, specifying 3-11
Files
.dbxinit 2-3
.o 1-16
core 3-18; 5-1
designating the current file 1-12
specifying source directories 5-2
Find pattern in source file 5-12
Foreground execution of process ..... 4-4 to 5
fork() system call 4-1
func command 3-23; 4-3; 5-35

Function
displaying value of 5-40

G-H

gripe command C-1
help command 3-1; 5-36
heterogeneous applications, debugging 4-1
Hexadecimal and octal
displaying addresses, constants, variables 5-14
representation 5-8
homogeneous parallel applications, debugging 4-1

I

I/O address space, precautions for accessing 3-1
Identifiers
clarifying scope of 5-83
containing 5-4
description of 5-4
disambiguating 5-5
listing context for 5-82
when to use back quotes 5-4
with duplicate names 3-16
context for interpreting 3-22
Identifying
a coredump file 5-2
files to debug 5-2
processes 1-6
ignore command 1-7; 3-24; 5-37
Ignoring signals or events 1-17; 3-24; 5-37
Include files, tracing in 5-76
Indexing arrays 3-4
Initializing Pdbx
on invocation 5-2
using the **–c** option 5-2
Invoking
Pdbx 2-5
Pdbx as dbx 3-26

J-K-L

Kernel code, executing 3-23
Killing orphan processes 1-3
Language considerations while debugging 3-4
Library routines, monitoring 2-2
Line numbers 5-6
Linked list, displaying a 5-56
Linking programs for used with Pdbx 2-3
list command 1-7; 3-9; 5-38
listi command 5-38
Listing
  a source file 5-38
  contexts for an identifier 5-82
  current aliases 1-9
  processes being debugged 5-49
Listings, program
  fac.c 1-4
  pfac.f 1-15
LOCAL 1-22
Logical (Boolean) expressions 5-7

M

m_fork() routine 4-1
Machine-level debugging 5-14,38
Memory, displaying in raw format 5-14
Memory-mapped devices, precautions for accessing 3-1
Messages
  `assuming parallel library used` 4-5
  error 3-2
  `expression too large to evaluate` 5-41
  fatal error 3-2
  `no executable code` 1-10
  stopped (tty input) C-2
  `syntax error` 3-2
  warning 3-2
See also Error messages
Multiline commands 5-4
Multiple commands in command line 5-3
Multiple executable files 5-2
Multiple-line statements 5-76

Multiple-program applications, debugging 4-6

N

next command 5-39
nexti command 5-39
`no executable code` message 1-10

O

On-line synopsis of commands 5-36
Operating procedures ..... Chapter 3
Optimizations 1-11
Ordering of command line options 5-2,3
Orphan process, stopping 3-25
Overview
  demonstration programs ..... 1-2 to 3
  Pdbx ..... 1-1 to 2

P

Parallel applications, debugging ..... Chapter 4
Parallel Programming Library 1-16; 4-5
Parameters of active procedures, displaying 5-81
Pattern search in source file 5-12
Pdbx
  command line options 5-2
  command line syntax 3-5
  debugging parallel applications 4-1
  enhancements to 4.3bsd dbx C-1
  exiting 3-8
  exiting Pdbx 5-51
  initializing 2-3
  invoked as dbx 3-26
  messages C-2
  overview ..... 1-1 to 2
  reserved words B-1
  suspending 3-21
  suspending Pdbx 5-71
pdbx command, described 5-1

Pdbx commands
% 5-16
/ 5-12
? 5-12
alias 5-17
assign 5-19
call 5-22
case of 5-4
catch 5-24
cont 5-26
create 5-28
delete 5-29
down 5-30
dump 5-32
edit 5-33
file 5-34
func 5-35
help 3-1; 5-36
ignore 5-37
list 5-38
listi 5-38
next 5-39
nexti 5-39
print 5-40
print fpustack 5-45
ps 5-49
quit 5-51
release 5-52
rerun 5-55
return 5-54
run 5-55
set 5-56
sh 5-60
signal 5-61
source 2-3; 5-63
status 5-64
step 5-65
stepi 5-65
stop 5-66,71
stopi 5-66,71
terminate 5-72
trace 5-73

tracei 5-73
unalias 5-17
unset 5-56
up 5-78
use 5-79
whatis 5-80
where 5-81
whereis 5-82
which 5-83
Physical address space, precautions for accessing 3-1
pmap device driver, precautions for using 3-1
Pointers 3-4
coercion of absolute address to pointer 5-9
Preserving case 5-3
print command 3-13; 5-40
displaying 80387 registers 5-45
displaying strings 5-21
print fpustack command 5-45
$fpustackall debugger variable 5-46
Printing
80387 registers 5-45
active stop commands 5-64
active trace commands 5-64
aliases for Pdbx commands 5-17
declaration for identifier 5-80
double-precision registers 5-42
single-precision registers 5-42
traceback from current procedure 5-81
value of a function 1-8
value of a variable or expression 5-40
variables in specified procedure 5-32
Procedure
execute selected procedure 5-22
listing active procedures 5-81
Process
current 4-2
identification 1-6
terminating selected processes 5-72
Processes
background execution of 4-4
created by Pdbx 1-6
creating a new process 5-28,55

foreground execution of 4-4
identifying 1-6
killing orphan processes 3-25
listing active processes 5-49
running 5-55
terminating 5-55
Program image 1-6; 3-7
Program listings
fac.c 1-4
pfac.f 1-15
Program, current 4-2
ps command 3-8; 5-49
use in determining where a process stopped 3-19
use when analyzing core dumps 3-18
ptrace() system call 3-23

Q-R

quit command 1-22; 3-8; 5-51
Radix, setting 3-20
Record syntax 3-4
Recursive procedure, selecting instance of 5-30
REDUCTION 1-22
Referencing arrays 3-4
Registers
assigning values to 5-19
displaying 80387 registers 5-45
displaying contents 5-40
names 5-8
printing double-precision 5-42
printing single-precision 5-42
release command 5-52
Release process stopped by fork 5-52
rerun command 1-7; 3-7; 5-55
Reserved words
list of B-1
using 3-2
Resolving identifier conflicts 3-16
Resuming program execution 5-26
return command 5-54
Returning temporarily to shell 5-71
Returning to a procedure 5-54

run command 1-6; 3-6; 5-55
Running
a process 5-26,55
a program 5-55
processes asynchronously 5-2
the debugger 2-5

S

Search for pattern in source file 5-12
Segmentation faults
catching 4-5
parallel program 5-24
Semicolon (;) in command line 5-3
Sending a signal to a process 5-61
set command 5-56
Setting
breakpoints 1-7; 3-8; 5-66
radix 3-20
the current source file 5-34
Setting up your debug environment 2-3
Setuid programs, referencing 3-5; 5-52
sh command 5-60
Shell
executing new shell from Pdbx 5-60
returning temporarily to shell from Pdbx 5-71
shell command 3-21
SIGINT 3-24
signal command 5-61
Signal, definition of 3-24
Signals C-1
as described in **signal**(2) 3-24
catching 5-24
cont command 5-26
delivery and suppression of caught signals 5-26
displaying ignored signals 5-37
ignoring 1-17
multiple signals pending 5-26
SIGINT 3-24
signal command 5-61
SIGSEGV 1-17; 4-5; 5-24
SIGSTOP 3-25

SIGTSTP 3-25
terminal signals ..... 3-25 to 26; 4-5
SIGSEGV signals
ignoring 1-17
use in parallel programs 4-5
SIGSTOP 3-25
SIGTSTP 3-25
Single-stepping 5-39,65
Size restrictions, source file 2-1
source command 2-3; 5-63
Source files
debugging large 2-1
editing 5-33
line numbers 5-6
specifying source directories 5-79
Spaces in command lines 5-4
Specifying
directories where source files reside 5-79
files to debug 5-2
the current procedure 5-35
the current source file 5-34
Specifying arrays
with brackets 3-4
with parentheses 3-4
Stack
displaying 80387 registers 5-45
Stack frame
selecting 5-30,35
Stack space in parallel programs 4-5
Statements, multiple line 5-76
status command 1-11; 3-8; 5-64
Status word 5-45
step command 5-65
stepi command 5-65
stop all command 3-18
stop command 3-8; 5-66
canceling 5-29
listing currently defined stops 5-64
use when debugging parallel applications 4-1
stopi command 5-66
Stopping a running process ..... 3-25 to 26
Stopping all processes 3-18

String search in source file 5-12
Strings 5-8
Struct syntax 3-4
suspend command 3-21; 5-71
Suspending pdbx 3-21
Symbol table 2-2
Synchronous and asynchronous terminal I/O ..... 4-4 to 5
Syntax
  for Pdbx commands 3-5
  of Pdbx commands ..... 5-3 to 11
`syntax error` message 3-2
sys.c listing C-8
System calls, monitoring 2-2

T

Tag word 5-45
Terminal I/O
  synchronous and asynchronous ..... 4-4 to 5
terminate command 3-7; 5-72
Terminating ..... selected processes
  a process 3-8; 5-55
  Pdbx 5-51
trace command 1-11; 3-15; 5-73
  canceling 5-29
  listing currently defined traces 5-64
Tracebacks 5-81
tracei command 5-73
Tracing in $INCLUDE files 5-76
Tracing program execution 3-14; 5-73
Traps 5-27
Type checking
  for **assign** command 5-19
  for **call** command 5-22
Type conversion 5-9

U

unalias command 5-17
UNIX 4.3bsd dbx compared to Pdbx ..... Appendix C
unset command 5-56
up command 3-22; 4-3; 5-78

use command 3-11; 5-79
`using memory image in core` message 2-7
Using Pdbx ..... Chapter 3

V

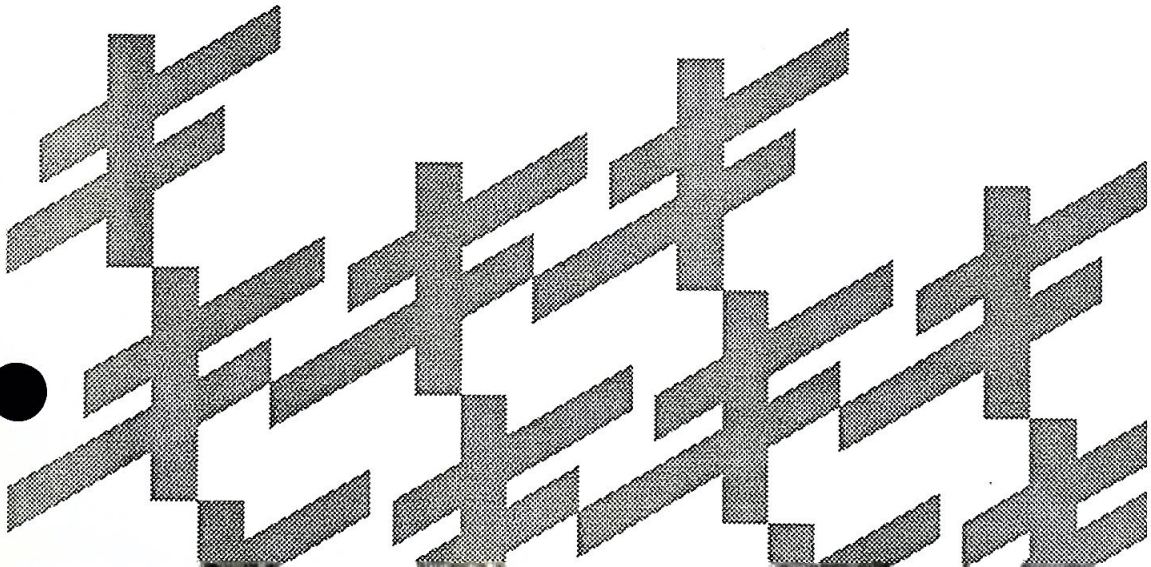
Variables
  assigning values to 5-19
  displaying value of 3-13; 5-32,40
    in decimal 5-14
    in hexadecimal 5-14
  LOCAL 1-22
  reduction 1-22
vfork, support of 2-1

W-X-Y-Z

Warning messages A-1
whatis command 3-16; 5-5,80
where command 3-19; 5-81
  use when analyzing core dumps 3-18
whereis command 3-16; 5-5,82
which command 3-16; 5-5,83

DYNIX/ptx®
C and Language Tools
Man Pages

1003-49195-00
Sequent Computer Systems, Inc.



DYNIX/ptx is a registered trademark of Sequent Computer Systems, Inc.

Symmetry and Pdbx are trademarks of Sequent Computer Systems, Inc.

IBM is a registered trademark of International Business Machines Corp.

UNIX is a registered trademark of AT&T.

Intel is a trademark of Intel Corporation.

Copyright 1989 by Sequent Computer Systems, Inc. All rights reserved. This document may not be copied or reproduced in any form without permission from Sequent Computer Systems, Inc. Information in this document is subject to change without notice.

Printed in the United States of America.

NAME

ar – maintain archives and libraries for portable archives

SYNOPSIS

ar key [posname] afile [name] ...

DESCRIPTION

The **ar**(1) command maintains groups of files combined into a single archive file. Its main use is to create and update library files as used by the link editor. It can be used, though, for any similar purpose. The magic string and the file headers used by **ar**(1) consist of printable ASCII characters. If an archive is composed of printable files, the entire archive is printable.

When **ar**(1) creates an archive, it creates headers in a format that is portable across all machines. The portable archive format and structure is described in detail in **ar**(4). The archive symbol table [described in **ar**(4)] is used by the link editor [**ld**(1)] to effect multiple passes over libraries of object files in an efficient manner. An archive symbol table is created and maintained by **ar**(1) only when there is at least one object file in the archive. The archive symbol table is in a specially named file that is always the first file in the archive. This file is never mentioned or accessible to the user. Whenever the **ar**(1) command is used to create or update the contents of such an archive, the symbol table is rebuilt. The **s** option described below will force the symbol table to be rebuilt when no other action is desired.

Unlike command options, the command key is a required part of the *ar* command line. The *key* (which may begin with a –) is formed with one of the following letters: **drqtpmx**. Arguments to the key, alternatively, are made with one or more of the following set: **vuaibcls**. *posname* is an archive member name used as a reference point in positioning other files in the archive. *afile* is the archive file. The *names* are constituent files in the archive file. The meanings of the *key* characters are as follows:

d Delete the named files from the archive file.

r Replace the named files in the archive file. If no optional positioning argument (**a b i**) is specified, new files are placed at the end.

q Quickly append the named files to the end of the archive file. Optional positioning characters are invalid. The command does not check whether the added members are already in the archive. This option is useful to avoid quadratic behavior when creating a large archive piece-by-piece. Unchecked, the file may grow exponentially up to the second degree.

t Print a table of contents of the archive file. If no names are given, all files in the archive are tabled. If names are given, only those files are tabled.

p Print the named files in the archive to the standard output.

m Move the named files to the end of the archive. If a positioning character is present, then the *posname* argument must be present and, as in **r**, specifies where the files are to be moved.

x Extract the named files. If no names are given, all files in the archive are extracted. In neither case does **x** alter the archive file.

The meanings of the *key* arguments are as follows:

a Place the named file after (**a**) *posname*. The *posname* argument must be present. **a** is used with the **r** key.

b Place the named file before (**b**) *posname*. The *posname* argument must be present. **b** is used with the **r** key.

i Place the named file before *posname*. The *posname* argument must be present. **i** is used with the **r** key.

v Give a verbose file-by-file description of the making of a new archive file from the old archive and the constituent files. When used with **t**, give a long listing of all information about the files. When used with **x**, precede each file with a name.

c Suppress the message that is produced by default when *afile* is created.

l Place temporary files in the local (current working) directory rather than in the default temporary directory, *TMPDIR*.

s Force the regeneration of the archive symbol table even if **ar**(1) is not invoked with a command that will modify the archive contents. This command is useful to restore the archive symbol table after the **strip**(1) command has been used on the archive.

u Update the archive files only if the modification dates of the new files are later than those of the archive files. This argument is used with the **r** key.

FILES

$TMPDIR/* Temporary files

$TMPDIR is usually */usr/tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

ld(1), lorder(1), strip(1), tmpnam(3S), a.out(4), ar(4)

NOTES

If the same file is mentioned twice in an argument list, it may be put in the archive twice.

File names are truncated to 14 characters when placed in an archive. Names longer than 14 characters will be truncated to 14 characters when used in an **extract** operation.

NAME

ar – common archive file format

SYNOPSIS

#include <ar.h>

DESCRIPTION

The archive command **ar**(1) is used to combine several files into one. Archives are used mainly as libraries to be searched by the link editor **ld**(1).

Each archive begins with the archive magic string. .nf

```
#define  ARMAG   "!<arch>\n"   /* magic string */ #define  SAR-
MAG  8              /* length of magic string */
```

Each archive which contains common object files [see **a.out**(4)] includes an archive symbol table. This symbol table is used by the link editor **ld**(1) to determine which archive members must be loaded during the link edit process. The archive symbol table (if it exists) is always the first file in the archive (but is never listed) and is automatically created and/or updated by **ar**.

Following the archive magic string are the archive file members. Each file member is preceded by a file member header which is of the following format:

```
#define  ARFMAG    "`\n" /* header trailer string */

struct  ar_hdr            /* file member header */
{
    char    ar_name[16]; /* '/' terminated file member name */
    char    ar_date[12]; /* file member date */
    char    ar_uid[6];   /* file member user identification */
    char    ar_gid[6];   /* file member group identification */
    char    ar_mode[8];  /* file member mode (octal) */
    char    ar_size[10]; /* file member size */
    char    ar_fmag[2];  /* header trailer string */
};
```

All information in the file member headers is in printable ASCII. The numeric information contained in the headers is stored as decimal numbers (except for **ar_mode** which is in octal). Thus, if the archive contains printable files, the archive itself is printable.

The **ar_name** field is blank-padded and slash (/) terminated. The **ar_date** field is the modification date of the file at the time of its insertion into the archive. Common format archives can be moved from system to system as long as the portable archive command **ar**(1) is used. Conversion tools such as **convert**(1) exist to aid in the transportation of non-common format archives to this format.

Each archive file member begins on an even-byte boundary; a newline is inserted between files if necessary. Nevertheless, the size given reflects the actual size of the file, exclusive of padding.

Notice there is no provision for empty areas in an archive file.

If the archive symbol table exists, the first file in the archive has a zero-length name (i.e., `ar_name[0] == '/'`). The contents of this file are as follows:

- The number of symbols. Length: 4 bytes.
- The array of offsets into the archive file. Length: `4 bytes * number_of_symbols`.
- The name string table. Length: `ar_size - (4 bytes * (number_of_symbols + 1))`.

The number of symbols and the array of offsets are managed with **sgetl** and **sputl**. The string table contains exactly as many null-terminated strings as there are elements in the offsets array. Each offset from the array is associated with the corresponding name from the string table (in order). The names in the string table are all the defined global symbols found in the common object files in the archive. Each offset is the location of the archive header for the associated symbol.

SEE ALSO

ar(1), ld(1), strip(1), sputl(3X), a.out(4)

WARNINGS

strip(1) removes all archive symbol entries from the header. The archive symbol entries must be restored via the **ts** option of the **ar**(1) command before the archive can be used with the link editor **ld**(1).

NAME

as – common assembler

SYNOPSIS

as [options] [filename]

DESCRIPTION

The **as**(1) command assembles the named file. If no file is specified, **as**(1) takes its input from the standard input and places the output in the *a.out* file. The following flags may be specified in any order:

–o *objfile*	Put the output of the assembly in *objfile*. By default, the output filename is formed by removing the *.s* suffix, if there is one, from the input filename and appending a *.o* suffix.
–r	Make data read only by merging the data section with the text section.
–k	Assemble for stand alone.
–m	Run the **m4**(1) macro processor.
–n	Turn off long/short address optimization. By default, address optimization takes place.
–R	Remove (unlink) the input file after assembly is completed.
–dl	Do not produce line number information in the object file.
–u	Remove unreferenced debugging symbols from the symbol table.
–V	Write the version number of the assembler being run on the standard error output.
–Y [*md*],*dir*	Find the **m4**(1) preprocessor (**m**) and/or the file of predefined macros (**d**) in directory *dir* instead of in the customary place.

FILES

TMPDIR/* Temporary files

TMPDIR is usually */tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

cc(1), ld(1), nm(1), strip(1), tmpnam(3S), a.out(4)

WARNING

If the **–m** (**m4**(1) macro processor invocation) option is used, keywords for **m4**(1) cannot be used as symbols (variables, functions, labels) in the input file because **m4**(1) cannot determine which are assembler symbols and which are real **m4**(1) macros.

NOTES

Arithmetic expressions may have only one forward referenced symbol per expression.

Wherever possible, the assembler should be accessed through a compilation system interface program [such as **cc**(1)].

NAME

cb – C program beautifier

SYNOPSIS

cb [**–s**] [**–j**] [**–l** leng] [file ...]

DESCRIPTION

The **cb**(1) comand reads C programs, either from its arguments or from the standard input, and writes them on the standard output with spacing and indentation that display the structure of the code. Under default options, **cb**(1) preserves all user newlines.

cb(1) accepts the following options.

–s Simplifies the code into the style of Kernighan and Ritchie in *The C Programming Language*.

–j Causes split lines to be put back together.

–l *leng* Causes **cb**(1) to split lines that are longer than *leng*.

SEE ALSO

cc(1)

BUGS

Punctuation that is hidden in preprocessor statements will cause indentation errors.

NAME

cc – C compiler

SYNOPSIS

cc [option] ... file ...

DESCRIPTION

The **cc**(1) command is used to compile and link C programs. **cc**(1) accepts several types of input files.

Files having names that end with *.c* indicate C source programs; they are compiled, producing files of the same name but with the suffix *.o* substituted for *.c*. The *.o* file is normally deleted if a single C program is compiled and linked at the same time.

Files having names that end with *.s* indicate assembly language source programs; they are assembled, producing *.o* files.

All other files are passed to **ld**(1) in the link phase. For more information on producing an executable file, refer to the **ld**(1) description in the language tools binder.

The C preprocessor options **–H, –D, –U, –I,** and **–C** are recognized on the command line and passed to the C preprocessor. Refer to **cpp**(1) for more detail about the C preprocessor.

cc(1) passes unrecognized options to **ld**(1). See **ld**(1) for link-time options. The following options are interpreted by **cc**(1):

–c	Suppress the linking phase of the compilation; save the object files.
–C	Prevent the **cpp**(1) macro preprocessor from removing comments from the file.
–dl	Suppress the generation of line number information for the debugger. This option is meaningful only when the **–g** option is specified.
–ds	Suppress the generation of symbol attribute information for the debugger. This option is meaningful only when the **–g** option is specified.
–D*name*[=*value*]	Define a name. Using **–D** has the same result as using a *#define* statement in your program. The option may be followed by a name and a value for the name. If no value is given, the value 1 is used.
–E	Run only the **cpp**(1) macro preprocessor on the named C programs, and send the result to the standard output.
–g	Produce additional symbol table information for **pdbx**(1) and pass the **–lg** option to **ld**(1). This option suppresses default optimizations performed by the compiler that may conflict with debugging.

–I*pathname* Search for *#include* files in *pathname* before searching in the default *INCDIR*. *pathname* must be a directory. This option applies only to non-absolute files (those files having names that do not begin with /). If the *#include* file name is enclosed in " ", **cpp**(1) searches first in the directory of the file with the *#include* line. If the file is not found, or if the file was enclosed in < >, **cpp**(1) then searches each *pathname* specified by a **–I** option. Finally, **cpp**(1) searches the default *INCDIR*. Up to eight **–I** options, each followed by a directory name, may be specified on the command line.

–o *filename* Name the final output object file *filename*. The default name of the final output object file is *a.out*. When this option is used and *filename* is not *a.out*, any existing *a.out* file is left undisturbed.

–O Optimize code. Same as the **–Wc,–O2** option.

–p Produce code for profiling the performance of the object program with **prof**(1). If the object modules are being linked, a profiled version of the library is searched first by the linker. When the object program is run, the file *mon.out* is created in the current directory. An execution profile can then be generated by using **prof**(1).

–P Run only **cpp**(1) on the named C programs and leave the result in corresponding files suffixed *.i*.

–S Compile and do not assemble the named C programs. Leave the assembler language output in corresponding files suffixed *.s*.

–U*name* Remove any initial definition of a predefined name.

–V Print the version number of the Sequent C compiler on standard output.

–W*x,arg1*[*,arg2...*] Pass the arguments specified by *arg1, arg2,* and so forth, which are any of the valid flags appropriate for the compiler pass specified by *x*, where *x* is one of the following:
p indicates *cpp,* the preprocessor
0 indicates *ccom,* the C compiler (note this is the number 0, not the letter O)
1 indicates *cgen,* the code generator (note this is the number 1, not the letter l)
i indicates *igen,* the inline code generator
a indicates *as,* the assembler
l indicates *ld,* the linker
c indicates the *cc* driver itself, as in **–Wc,–fpa**

-Yc,*pathname* Specify a new directory, *pathname*, for the location of the tools designated by *c*, where *c* is one of the following:
p indicates *cpp*, the preprocessor
0 indicates *ccom*, the C compiler (note this is the number 0, not the letter O)
1 indicates *cgen*, the code generator (note this is the number 1, not the letter l)
i indicates *igen*, the inline code generator
a indicates *as*, the assembler
l indicates *ld*, the linker
S indicates the start-up routine directory (where *crt0.o* resides)
I indicates the **cpp**(1) default include directory (defaults to */usr/include)*
L indicates the first **ld**(1) default library directory (defaults to */lib)*
U indicates the second **ld**(1) default library directory (defaults to */usr/lib)*

-# List on standard error the utilities and their arguments called by *cc*. This information is useful for debugging unexpected *cc* behavior.

-## Provide the same function as **-#** , plus list the full pathname of the utility that will be executed.

-### Provide the same function as **-##** except that no execution is performed.

-Wc,-fpa Produce code for the Weitek 1167 floating-point accelerator. By default, the C compiler produces code for the Intel 80387 floating-point co-processor. The **-fpa** option is also passed to the linker, **ld**(1), which selects the Weitek versions of the C run-time libraries. The linker issues a warning if non **-fpa** (Intel 80387) code and **-fpa** (Weitek 1167) code are linked together. To produce correct results, a **-fpa** routine must not return a value to a non **-fpa** routine and vice-versa.

-Wc,-fpa also defines the preprocessor symbol **w1167** for use in conditional compilation. When **-Wc,-fpa** is not specified, the preprocessor symbol **i387** is defined.

When the **-fpa** option is used, the C compiler will generate Intel 80387 code in those cases when the Intel 80387 processor is faster than the Weitek 1167 processor. This special use of the Intel 80387 processor is transparent to the programmer and does not affect the issue of mixing **-fpa** and non **-fpa** code.

-Wc,-seq Recognize the Sequent keywords *const*, *volatile*, *signed*, *shared*, and *private*. By default the compiler does not recognize these keywords. Refer to the *DYNIX/ptx C User's Manual* for more information about using these keywords. Refer to the *Guide to Parallel Programming* for information about the parallel programming extensions to the C language that are supported by the C compiler.

-Wc,-Y Treat variables as shared by default, rather than private.

-Wc,-i Treat all variables as if they were declared *volatile*. This option can be used to correctly compile programs that should have *volatile* on some variable declarations but do not. This option suppresses many of the default optimizations the compiler otherwise performs. This option also forces the compiler to save and restore values in memory. These values could otherwise be placed in registers. Typically this option produces worse code than the **-Wc,-Og** option.

-Wc,-Og Provide minimum optimization. This option specifies the same optimization level as **-g**. However, line numbers and symbol information are not generated.

-Wc,-Osz Optimize code for smallest code size.

-Wc,-O0 Provide default code optimization. Optimizations that would increase code size, such as inlining, or that require significant additional compilation time are not performed. Increasing the optimization level generally improves program speed, but it is not guaranteed to do so. Such improvement depends on the code.

-Wc,-O1 Provide the same level of code optimization as provided by the **-Wc,-O0** option plus invariant code motion, strength reduction, and code duplication. No function inlining is performed.

-Wc,-O2 Provide the same level of code optimization as provided by the **-Wc,-O1** option plus inlining of small user-defined functions found in the same *.c* file.

-Wc,-O3 Provide the same level of code optimization as provided by the **-Wc,-O2** option plus inlining of certain library functions. If the target processor is an Intel 80387, inline code specific to this processor is generated for these *libm* routines: *sin*, *cos*, *tan*, *atan*, *atan2*, and *sqrt*. The inline code for these routines does not check the arguments passed to the routines for a possible exception. If an exception occurs, a floating-point exception trap is generated. When inlining does not occur, this exception trap results in *matherr* being called. However, the inline code does not make this call. For the *fabs* routine, inline code for either the Intel 80387 or Weitek 1167 processor (as selected by **-Wc,-fpa**) is generated. Neither the inline code for *fabs* nor the *libm fabs* routine perform argument checking.

memset and *memcpy* routines in *libc* can also be affected by the optimization level. The inline version of *memset* is functionally identical to the *libc* version of the routine. The inline version of *memcpy* differs from the *libc* version in how overlap conditions are checked. For example, if the program depends on the first element of the array always being copied to the second element of the array, the inline version may provide a different result than is expected.

The criteria for inlining fuctions are many and varied. These criteria depend upon compile time memory usage, types of the arguments, and similar considerations. These criteria may vary from release to release. Refer to the *DYNIX/ptx C User's Manual* for more information about optimization.

-Wc,-SO Send the assembly language output files to standard ouptut. The named C programs are compiled and the *.s* files containing the assembly language output files are sent to standard output.

FILES

file.c	input file
file.o	object file
file.s	assembly file
a.out	linked output

/usr/xltr/c/<version>/lib/

ccom	C compiler
cgen	code generator
igen	inline code generator

/lib/cpp	preprocessor
/lib/crt0.o	runtime startoff
/lib/mcrt0.o	startoff for profiling
/lib/libc.a	standard library, see *intro*(3)
/usr/lib/libp/libc.a	profiling library, see *intro*(3)
/usr/lib/libpps.a	parallel prog library, see *intro*(3)
/usr/lib/libseq.a	Sequent-specific library
/usr/include	standard directory for *#include* files
mon.out	file produced for analysis by **prof**(1)
INCDIR	*#include* files dir list, (*/usr/include*)
TMPDIR/*	Temporary files

Temporary files are usually placed in */tmp*, but this behavior can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

monitor(3c), prof(1), ld(1), pdbx(1), as(1)

NOTES

When the **-Wc,-eeq** option is used, the Sequent C compiler treats the words *const, private, signed, shared,* and *volatile* as keywords, and these words cannot be used as identifiers. This occasionally causes confusion when code containing these identifiers is ported from other systems.

When the **-c** flag is specified, **cc**(1) accepts invalid options or filenames without producing warnings. Such behavior occurs because unrecognized options and files are saved for the link phase, but when **-c** is specified, the link phase is never run.

NAME

cflow – generate C flowgraph

SYNOPSIS

cflow [**–r**] [**–ix**] [**–i_**] [**–d**num] files

DESCRIPTION

The **cflow** command analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed with *.y*, *.l*, and *.c* are yacced, lexed, and C-preprocessed as appropriate. The results of the preprocessed files and files suffixed with *.i* are then run through the first pass of **lint**(1). Files suffixed with *.s* are assembled. Assembled files and files suffixed with *.o* have information extracted from their symbol tables. The results are collected and turned into a graph of external references which is displayed on the standard output.

Each line of output begins with a reference number, followed by a suitable number of tabs indicating the level, then the name of the global symbol, followed by a colon and its definition. Normally, only function names that do not begin with an underscore are listed (see the **–i** options below). For information extracted from C source, the definition consists of an abstract type declaration (e.g., **char** *) and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only **<>** is printed.

As an example, given the following in *file.c*:

```
int    i;

main()
{
        f();
        g();
        f();
}

f()
{
        i = h();
}
```

the command **cflow -ix file.c** produces this output:

```
1     main: int(), <file.c 4>
2           f: int(), <file.c 11>
3                 h: <>
4                 i: int, <file.c 1>
5           g: <>
```

When the nesting level becomes too deep, the output of **cflow**(1) can be piped to **pr**(1), using the **-e** option, to compress the tab expansion to something less than every eight spaces.

In addition to the **-D**, **-I**, and **-U** options [which are interpreted just as they are by **cc**(1) and **cpp**(1)], the following options are interpreted by **cflow**(1):

-r Reverse the "caller:callee" relationship, producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

-ix Include external and static data symbols. The default is to include only functions in the flowgraph.

-i_ Include names that begin with an underscore. The default is to exclude these functions (and data if **-ix** is used).

-d*num* The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a nonpositive integer will be ignored.

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), pr(1), yacc(1)

DIAGNOSTICS

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (e.g., the C-preprocessor).

BUGS

Files produced by **lex**(1) and **yacc**(1) cause the reordering of line number declarations which can confuse **cflow**(1). To get proper results, feed **cflow**(1) the **yacc**(1) or **lex**(1) input.

NAME

cpp – the C language preprocessor

SYNOPSIS

LIBDIR/cpp [option ...] [ifile [ofile]]

DESCRIPTION

The C language preprocessor, **cpp**(1), is invoked as the first pass of any C compilation by the **cc**(1) command. Thus, the output of **cpp**(1) is designed to be in a form acceptable as input to the next pass of the C compiler. As the C language evolves, **cpp**(1) and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of **cpp**(1) other than through the **cc**(1) command is not suggested, because the functionality of **cpp**(1) may someday be moved elsewhere. See **m4**(1) for a general macro processor.

cpp(1) optionally accepts two filenames as arguments: *ifile* and *ofile* are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following options to **cpp**(1) are recognized:

–P Preprocess the input without producing the line control information used by the next pass of the C compiler.

–C By default, **cpp**(1) strips C-style comments. If the **–C** option is specified, all comments (except those found on **cpp**(1) directive lines) are passed along.

–U*name* Remove any initial definition of *name*. The following list shows the current symbols.

operating system:	unix
hardware:	i386
UNIX system variant:	_SEQUENT_
lint(1):	lint

–D*name*

or

–D*name=def* Define *name* with value *def* as if by a *#define* statement. If no *=def* is given, *name* is defined with value 1. The **–D** option has lower precedence than the **–U** option. That is, if the same name is used in both a **–U** option and a **–D** option, the name will be undefined regardless of the order of the options.

–T The **–T** option forces **cpp**(1) to use only the first eight characters to distinguish preprocessor symbols and is included for backward compatibility.

–I*dir* Search for *#include* files in *dir* before searching in the default *INCDIR*. *dir* must be a directory. This option applies only to non-absolute files (those files having names that do not begin with /). If the *#include* file name is enclosed in " ", **cpp**(1) searches first in the directory of the file with the *#include* line. If the file is not found, or if the file was enclosed in <>, **cpp**(1) then searches each *dir* specified by a **–I** option. Finally, **cpp**(1) searches the default *INCDIR*. Up to eight **–I** options, each followed by a directory name, may be specified on the command line.

–Y*dir* Use directory *dir* in place of the standard directory *(INCDIR)* when searching for *#include* files.

–H Print, one per line on standard error, the pathnames of included files.

Two special names are understood by **cpp**(1). The name __LINE__ is defined as the current line number (expressed as a decimal integer) as known by **cpp**(1), and __FILE__ is defined as the current filename (expressed as a C string) as known by **cpp(1)**. They can be used anywhere (including in macros) just as any other defined name.

All **cpp**(1) directive lines start with # in column 1. Any number of blanks and tabs are allowed between the # and the directive. The directives are as follows:

#define *name token-string*
Replace subsequent instances of *arg* with *token-string*.

#define *name*(*arg*, ..., *arg*) *token-string*
Notice that there can be no space between *name* and the left parenthesis (. Replace subsequent instances of *name* followed by a left parenthesis (, a list of comma-separated set of tokens, and a right parenthesis) followed by *token-string*, where each occurrence of an *arg* that appears in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, **cpp**(1) restarts its scan for names to expand at the beginning of the newly created *token-string*.

#undef *name*
Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the directive line after *name*.

#ident *"string"*
Put *string* into the .comment section of an object file.

#include *"filename"*

or

#include *<filename>*

Include at this point the contents of *filename* (which will then be run through **cpp**(1)). When the *<filename>* notation is used, *filename* is searched for only in the standard places. See the preceding **–I** and **–Y** options for more detail. No additional tokens are permitted on the directive line after the final " or >.

#line *integer-constant "filename"*

Causes **cpp**(1) to generate line-control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file from which it comes. If *"filename"* is not given, the current filename is unchanged. No additional tokens are permitted on the directive line after the optional *filename*.

#endif Ends a section of lines begun by a test directive (*#if*, *#if defined*, *#ifdef*, or *#ifndef*). Each test directive must have a matching *#endif*. No additional tokens are permitted on the directive line.

#ifdef *name* The lines following will appear in the output if and only if *name* has been the subject of a previous *#define* without being the subject of an intervening *#undef*. No additional tokens are permitted on the directive line after *name*.

#ifndef *name*

The lines following will appear in the output if and only if *name* has not been the subject of a previous *#define*. No additional tokens are permitted on the directive line after *name*.

#if *constant-expression*

Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, such as the **?:** operator, and the unary **–**, **!**, and **~** operators are legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator, *defined*, which can be used in *constant-expression* in these two forms: *defined* (*name*) or *defined name*. This allows the use of *#ifdef* and *#ifndef* in an *#if* directive. Only these operators, integer constants, and names which are known by **cpp**(1) should be used in *constant-expression*. In particular, the *sizeof* operator is not available.

To test whether either of two symbols, *foo* and *fum*, are defined, use the following:

```
#if defined(foo) || defined(fum)
```

#if defined (*const*)

The following lines will appear in the output if and only if the constant specified by *const* has been defined.

#elif *constant-expression*
An arbitrary number of *#elif* directives is allowed between a *#if*, *#ifdef*, or *#ifndef* directive and an *#else* or *#endif* directive. The lines following the *#elif* directive will appear in the output if and only if the preceding test directive evaluates to zero, all intervening *#elif* directives evaluate to zero, and the *constant-expression* evaluates to non-zero. If *constant-expression* evaluates to non-zero, all succeeding *#elif* and *#else* directives will be ignored. Any *constant-expression* allowed in an *#if* directive is allowed in an *#elif* directive.

#else The lines following will appear in the output if and only if the preceding test directive evaluates to zero and all intervening *#elif* directives evaluate to zero. No additional tokens are permitted on the directive line.

The test directives and the possible *#else* directives can be nested.

FILES

INCDIR	Standard directory list for *#include* files, usually */usr/include*
LIBDIR	Usually */lib*

SEE ALSO
cc(1), lint(1), m4(1)

DIAGNOSTICS
The error messages produced by **cpp**(1) are intended to be self-explanatory. The line number and filename where the error occurred are printed along with the diagnostic.

NOTES
The unsupported **-W** option enables the *#class* directive. If it encounters a *#class* directive, **cpp**(1) will exit with code 27 after finishing all other processing. This option provides support for "C with classes."

Because the standard directory for included files may be different in different environments, use this form of *#include* directive:

```
#include <file.h>
```

rather than one with an absolute path, like:

```
#include "/usr/include/file.h"
```

cpp(1) warns about the use of the absolute pathname.

NAME

cprs – compress a common object file

SYNOPSIS

cprs [–p] file1 file2

DESCRIPTION

The **cprs**(1) command reduces the size of a common object file, *file1*, by removing duplicate structure and union descriptors. The reduced file, *file2*, is produced as output.

cprs(1) is usually useful only on files that were compiled with the **–g** option.

cprs(1) has one option:

–p Print statistical messages including total number of tags, total duplicate tags, and total reduction of *file1*.

SEE ALSO

strip(1), a.out(4), syms(4)

NAME

ctrace – C program debugger

SYNOPSIS

ctrace [options] [file]

DESCRIPTION

The **ctrace**(1) command allows you to follow the execution of a C program, statement-by-statement. The effect is similar to executing a shell procedure with the **–x** option. **ctrace**(1) reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of **ctrace**(1) into a temporary file because the **cc**(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes, it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output so you can put it into a file for examination with an editor or the **bfs**(1) or **tail**(1) commands.

The options commonly used are as follows:

–f *functions* Trace only these *functions.*

–v *functions* Trace all but these *functions.*

You may want to add to the default formats for printing variables. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. Char, short, and int variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation. You can request that variables be printed in additional formats, if appropriate, with these options:

–o Octal

–x Hexadecimal

–u Unsigned

–e Floating point

The following options are used only in special circumstances:

–l *n* Check *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

–s Suppress redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

–t *n* Trace *n* variables per statement instead of the default of 10 (the maximum number is 20). The Diagnostics section explains when to use this option.

–P Run the C preprocessor on the input before tracing it. You can also use the **–D**, **–I**, and **–U** **cpp**(1) options.

The following options are used to tailor the run-time trace package when the traced program will run in a non-DYNIX system environment:

–b Use only basic functions in the trace code, that is, those in *ctype*(3C), *printf*(3S), and *string*(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have *signal*(2), *fflush*(3S), *longjmp*(3C), or *setjmp*(3C).

–p *string*
Change the trace print function from the default of 'printf('. For example, 'fprintf(stderr,' would send the trace to the standard error output.

–r *f* Use file *f* in place of the *runtime.c* trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the **–p** option).

EXAMPLE

Suppose the file *lc.c* contains this C program:

```
 1 #include <stdio.h>
 2 main()    /* count lines in input */
 3 {
 4     int c, nl;
 5
 6     nl = 0;
 7     while ((c = getchar()) != EOF)
 8           if (c = '\n')
 9                 ++nl;
10     printf("%d\n", nl);
11 }
```

To compile and execute the program, enter these commands and test data:

```
cc lc.c
a.out
1
(Ctrl-D)
```

The output of the program will be the number 2, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke **ctrace**(1) with these commands:

ctrace lc.c >temp.c
cc temp.c
a.out
the output will be as follows:

```
2 main()
6       nl = 0;
        /* nl == 0 */
7       while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be as follows:

```
        /* c == 49 or '1' */
8           if (c = '\n')
            /* c == 10 or '\n' */
9               ++nl;
                /* nl == 1 */
7       while ((c = getchar()) != EOF)
        /* c == 10 or '\n' */
8           if (c = '\n')
            /* c == 10 or '\n' */
9               ++nl;
                /* nl == 2 */
7       while ((c = getchar()) != EOF)
```

If you now enter an end-of-file character (Ctrl-D) the final output will be as follows:

```
        /* c == -1 */
10      printf("%d\n", nl);
        /* nl == 2 */2
        return
```

Note that the program output printed at the end of the trace line for the `nl` variable. Also note the `return` comment added by **ctrace**(1) at the end of the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable `c` is assigned the value '1' in line 7, but in line 8 it has the value '\n'. Once your attention is drawn to this **if** statement, you will probably realize that you used the assignment operator (=) in place of the equality operator (==). You can easily miss this error during code reading.

Execution-Time Trace Control

The default operation for **ctrace**(1) is to trace the entire program file, unless you use the **–f** or **–v** options to trace specific functions. This does not give you statement-by-statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding the **ctroff**() and **ctron**() function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with **if** statements, and you can even conditionally include this code because **ctrace**(1) defines the CTRACE preprocessor variable. For example:

```
#ifdef CTRACE
        if (c == '!' && i > 1000)
                ctron();
#endif
```

FILES

/usr/lib/ctrace/runtime.c Run-time trace package

SEE ALSO

bfs(1), tail(1), signal(2), ctype(3C), fclose(3S), printf(3S), setjmp(3C), string(3C)

DIAGNOSTICS

This section contains diagnostic messages from both **ctrace**(1) and **cc**(1), since the traced code often gets some **cc**(1) warning messages. You can get **cc**(1) error messages in some rare cases, all of which can be avoided.

ctrace Diagnostics

warning: some variables are not traced in this statement

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the **–t** option to increase this number.

warning: statement too long to trace

This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

cannot handle preprocessor code, use –P option

This is usually caused by #ifdef/#endif preprocessor statements in the middle of a C statement, or by a semicolon at the end of a #define preprocessor statement.

'if ... else if' sequence too long

Split the sequence by removing an **else** from the middle.

possible syntax error, try –P option

Use the **–P** option to preprocess the **ctrace**(1) input, along with any appropriate **–D**, **–I**, and **–U** preprocessor options. If you still get the error message, check the **WARNINGS** section below.

cc Diagnostics

warning: illegal combination of pointer and integer
warning: statement not reached
warning: sizeof returns 0

Ignore these messages.

compiler takes size of function

See the **ctrace**(1) "possible syntax error" message above.

yacc stack overflow

See the **ctrace**(1) "if ... else if' sequence too long" message above.

out of tree space; simplify expression

Use the **-t** option to reduce the number of traced variables per statement from the default of 10. Ignore the "ctrace: too many variables to trace" warnings you will now get.

redeclaration of signal

Either correct this declaration of *signal*(2), or remove it and #include <signal.h>.

WARNINGS

You will get a **ctrace**(1) syntax error if you omit the semicolon at the end of the last element declaration in a structure or union just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. Just use a different name.

ctrace(1) assumes that BADMAG is a preprocessor macro and that EOF and NULL are #defined constants. Declaring any of these to be variables, e.g., "int EOF;", will cause a syntax error.

BUGS

ctrace(1) does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all of the components of an aggregate when an assignment is made to the entire aggregate. **ctrace**(1) may choose to print the address of an aggregate or to use the wrong format (e.g., 3.149050e-311 for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

NAME

cxref – generate C program cross-reference

SYNOPSIS

cxref [options] files

DESCRIPTION

The **cxref**(1) command analyzes a collection of C files and attempts to build a cross-reference table. **cxref**(1) uses a special version of **cpp**(1) to include *#define*'d information in its symbol table. It produces a listing on standard output of all symbols (auto, static, and global) in each file separately, or, with the **–c** option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the **–D**, **–I** and **–U** options [which are interpreted just as they are by **cc**(1) and **cpp**(1)], the following options are interpreted by **cxref**(1):

–c Print a combined cross-reference of all input files.

–w*<num>* Width option which formats output no wider than <num> (decimal) columns. This option will default to 80 if <num> is not specified or is less than 51.

–o file Direct output to *file*.

–s Operate silently; do not print input filenames.

–t Format listing for 80-column width.

FILES

LLIBDIR Usually */usr/lib*

LLIBDIR/xcpp Special version of the C preprocessor

SEE ALSO

cc(1), cpp(1)

DIAGNOSTICS

Error messages are unusually cryptic, but usually mean that you cannot compile these files.

BUGS

cxref(1) considers a formal argument in a *#define* macro definition to be a declaration of that symbol. For example, a program that *#includes* **ctype.h** will contain many declarations of the variable *c*.

NAME

dis – object code disassembler

SYNOPSIS

dis [**–o**] [**–V**] [**–L**] [**–s**] [**–d** sec] [**–da** sec] [**–F** function] [**–t** sec] [**–l** string] file ...

DESCRIPTION

The **dis**(1) command produces an assembly language listing of *file*, which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following options are interpreted by the disassembler and may be specified in any order.

–o Print numbers in octal. The default is hexadecimal.

–V Print, on standard error, the version number of the disassembler being executed.

–L Look up source labels in the symbol table for subsequent printing. This option works only if the file was compiled with additional debugging information [e.g., the **–g** option of **cc**(1)].

–s Perform symbolic disassembly (i.e., specify source symbol names for operands where possible). Symbolic disassembly output will appear on the line following the instruction. For maximal symbolic disassembly to be performed, the file must be compiled with additional debugging information [e.g., the **–g** option of **cc**(1)]. Symbol names will be printed using C syntax.

–d *sec* Disassemble the named section as data, printing the offset of the data from the beginning of the section.

–da *sec* Disassemble the named section as data, printing the actual address of the data.

–F *function* Disassemble only the named function in each object file specified on the command line. The **–F** option may be specified multiple times on the command line.

–t *sec* Disassemble the named section as text.

–l *string* Disassemble the library file specified by *string*. For example, one would issue the command **dis –l x –l z** to disassemble *libx.a* and *libz.a*. All libraries are assumed to be in *LIBDIR*.

If the **–d**, **–da** or **–t** options are specified, only those named sections from each user-supplied filename will be disassembled. Otherwise, all sections containing text will be disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as [5], represents that the break-pointable line number starts with the following instruction. These line numbers will be printed only if the file was compiled with additional debugging information [e.g., the **-g** option of **cc**(1)]. An expression such as <40> in the operand field or in the symbolic disassembly, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A function name will appear in the first column, followed by ().

FILES

LIBDIR	Usually */lib*

SEE ALSO

as(1), cc(1), ld(1), a.out(4)

DIAGNOSTICS

The self-explanatory diagnostics indicate errors in the command line or problems encountered with the specified files.

NAME

dump – dump selected parts of an object file

SYNOPSIS

dump [options] files

DESCRIPTION

The **dump**(1) command dumps selected parts of each of its object *file* arguments.

This command will accept both object files and archives of object files. It processes each file argument according to one or more of the following options:

–a	Dump the archive header of each member of each archive file argument.
–g	Dump the global symbols in the symbol table of an archive.
–f	Dump each file header.
–o	Dump each optional header.
–h	Dump section headers.
–s	Dump section contents.
–r	Dump relocation information.
–l	Dump line number information.
–t	Dump symbol table entries.
–z *name*	Dump line number entries for the named function.
–c	Dump the string table.
–L	Interpret and print the contents of the *.lib* sections.

The following *modifiers* are used in conjunction with the options listed above to modify their capabilities.

–d *number*	Dump the section number, *number*, or the range of sections starting at *number* and ending at the *number* specified by **+d**.
+d *number*	Dump sections in the range beginning either with the first section or the section specified by **–d**.
–n *name*	Dump information pertaining only to the named entity. This *modifier* applies to **–h**, **–s**, **–r**, **–l**, and **–t**.
–p	Suppress printing of the headers.
–t *index*	Dump only the indexed symbol table entry. The **–t**, used in conjunction with **+t**, specifies a range of symbol table entries.
+t *index*	Dump the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the **–t** option.
–u	Underline the name of the file for emphasis.
–v	Dump information in symbolic representation rather than numeric (e.g., C_STATIC instead of 0X02). This *modifier* can be used with all the above options except **–s** and **–o** options of *dump*.

-z *name,number*
Dump line number entry or range of line numbers starting at *number* for the named function.

+z *number* Dump line numbers starting at either function *name* or *number* specified by **-z**, up to *number* specified by **+z**.

Blanks separating an *option* and its *modifier* are optional. The comma separating the name from the number modifying the **-z** option may be replaced by a blank.

The **dump**(1) command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal, or decimal representation as appropriate.

SEE ALSO

a.out(4), ar(4)

NAME

ld – link editor for common object files

SYNOPSIS

ld [options] filename

DESCRIPTION

The **ld**(1) command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and **ld**(1) combines the objects, producing an object module that can either be executed or, if the **–r** option is specified, used as input for a subsequent **ld**(1) run. The output of **ld**(1) is left in *a.out*. By default this file is executable if no errors occurred during the load. If any input file, *filename*, is not an object file, **ld**(1) assumes it is either an archive library or a text file containing link editor directives.

If any argument is a library, it is searched exactly once at the point it is encountered in the argument list. The library may be a relocatable archive library. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table [see **ar**(4)] is searched sequentially with as many passes as are necessary to resolve external references which can be satisfied by library members. Thus, the ordering of library members is functionally unimportant, unless multiple library members defining the same external symbol exist.

The following options are recognized by **ld**(1):

–e *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*.

–f *fill* Set the default fill pattern for "holes" within an output section as well as initialized *bss* sections. The argument *fill* is a two-byte constant.

–lx Search a library *libx.a*, where *x* is up to nine characters. A library is searched when its name is encountered, so the placement of a **–l** is significant. By default, libraries are located in */lib* or */usr/lib*.

–m Produce a map or listing of the input/output sections on the standard output.

–o *outfile* Produce an output object file by the name *outfile*. If no output file is specified, the output file defaults to *a.out*.

–r Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent **ld**(1) run. The link editor will not complain about unresolved references, and the output file will not be executable.

–a Create an absolute file. This is the default if the **–r** option is not used. Used with the **–r** option, **–a** allocates memory for common symbols.

–s Strip line number entries and symbol table information from the output object file.

-t Turn off the warning about multiply-defined symbols that are not the same size.

-u *symname* Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine. The placement of this option on the **ld**(1) line is significant; it must be placed before the library which will define the symbol.

-x Do not preserve local symbols in the output symbol table; enter external and static symbols only. This option saves some space in the output file. Using this option precludes the use of **pdbx**(1).

-z Do not bind anything to address zero. This option will allow runtime detection of null pointers.

-L *dir* Change the algorithm of searching for *libx.a* to look in *dir* before looking in *LIBDIR* and *LLIBDIR*. This option is effective only if it precedes the **-l** option on the command line.

-M Output a message for each multiply-defined external definition.

-N Put the text section at the beginning of the text segment rather than after all header information, and put the data section immediately following text in the core image.

-V Output a message giving information about the version of **ld**(1) being used.

-VS *num* Use *num* as a decimal version stamp identifying the *a.out* file that is produced. The version stamp is stored in the optional header.

-Y*[LU],dir* Change the default directory used for finding libraries. If **L** is specified, the first default directory which **ld**(1) searches, *LIBDIR*, is replaced by *dir*. If **U** is specified and **ld**(1) has been built with a second default directory, *LLIBDIR*, then that directory is replaced by *dir*. If **ld**(1) was built with only one default directory and **U** is specified, a warning is printed and the option is ignored.

FILES

LIBDIR/libx.a	Libraries
LLIBDIR/libx.a	Libraries
a.out	Output file
LIBDIR	Usually */lib*
LLIBDIR	Usually */usr/lib*

SEE ALSO

as(1), cc(1), exit(2), end(3C), a.out(4), ar(4)

WARNINGS

Through its options and input directives, the common link editor gives users great flexibility; however, those who use the input directives must assume some added responsibilities. Input directives and options should insure the following properties for programs:

- C defines a zero pointer as null. A pointer to which zero has been assigned must not point to any object. To satisfy this, users must not place any object at virtual address zero in the program's address space.
- When the link editor is called through **cc**(1), a startup routine is linked with the user's program. This routine calls **exit( )** [see **exit**(2)] after execution of the main program. If the user calls the link editor directly, then the user must insure that the program always calls **exit( )** rather than falling through the end of the entry routine.
- The default behavior of **ld**(1) is compatible with **exec**(2). If link editor directives are used, make sure that the resulting executable file is compatible with **exec**(2).

The symbols **etext**, **edata**, and **end** [see **end**(3C)] are reserved and are defined by the link editor. It is incorrect for a user program to redefine them.

If the link editor does not recognize an input file as an object file or an archive file, it will assume that it contains link editor directives and will attempt to parse it. This will occasionally produce an error message complaining about "syntax errors."

Arithmetic expressions may only have one forward-referenced symbol per expression.

NAME

lint – a C program checker

SYNOPSIS

lint [option] ... file ...

DESCRIPTION

The **lint**(1) command attempts to detect features of C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compiler. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

Arguments whose names end with *.c* are taken to be C source files. Arguments whose names end with *.ln* are taken to be the result of an earlier invocation of **lint**(1) with either the **–c** or the **–o** option used. The *.ln* files are analogous to *.o* (object) files that are produced by the **cc**(1) command when given a *.c* file as input. Files with other suffixes are warned about and ignored.

lint(1) will take all the *.c*, *.ln*, and (specified by **–lx**) files and process them in their command line order. By default, **lint**(1) appends the standard C lint library (*llib-lc.ln*) to the end of the list of files. However, if the **–p** option is used, the portable C lint library (*llib-port.ln*) is appended instead. When the **–c** option is not used, the second pass of **lint**(1) checks this list of files for mutual compatibility. When the **–c** option is used, the *.ln* and the *llib-lx.ln* files are ignored.

Any number of **lint**(1) options may be used, in any order, intermixed with filename arguments. The following options are used to suppress certain kinds of complaints:

–a Suppress complaints about assignments of long values to variables that are not long.

–b Suppress complaints about **break** statements that cannot be reached. (Programs produced by **lex** or **yacc** will often result in many such complaints).

–h Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

–u Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running **lint**(1) on a subset of files of a larger program).

–v Suppress complaints about unused arguments in functions.

–x Do not report variables referenced by external declarations but never used.

The following arguments alter the behavior of **lint**(1):

–lx Include additional lint library *llib-lx.ln*. For example, you can include a **lint**(1) version of the math library *llib-lm.ln* by inserting **–lm** on the command line. This argument does not suppress the default use of *llib-lc.ln*. These lint libraries must be in the assumed directory. This option can be used to reference local lint libraries and is useful in the development of multi-file projects.

–n Do not check compatibility against either the standard or the portable lint library.

–p Attempt to check portability to other dialects (IBM and GCOS) of C. Along with stricter checking, this option causes all non-external names to be truncated to eight characters and all external names to be truncated to six characters and one case.

–c Cause **lint**(1) to produce a *.ln* file for every *.c* file on the command line. These *.ln* files are the product of *lint*'s first pass only, and are not checked for inter-function compatibility.

–o x Cause **lint**(1) to create a lint library with the name *llib-lx.ln*. The **–c** option nullifies any use of the **–o** option. The lint library produced is the input that is given to *lint*'s second pass. The **–o** option simply causes this file to be saved in the named lint library. To produce a *llib-lx.ln* without extraneous messages, use of the **–x** option is suggested. The **–v** option is useful if the source file(s) for the lint library are just external interfaces (for example, the way the file *llib-lc* is written). These option settings are also available through the use of "lint comments" (see below).

–Wc,–seq
Recognize the Sequent keywords: *const, volatile, signed, shared,* and *private.*

–Wc,–fpa
lint(1) code intended for the floating-point accelerator. This option has an effect only if the **cpp**(1) macros **i387** or **w1167** were used for conditional compilation.

The **–D**, **–U**, and **–I** options of **cpp**(1) and the **–g** and **–O** options of **cc**(1) are also recognized as separate arguments. The **–g** and **–O** options are ignored, but, by recognizing these options, *lint*'s behavior is closer to that of the **cc**(1) command. Other options are warned about and ignored. The pre-processor symbol "lint" is defined to allow certain questionable code to be altered or removed for **lint**(1). Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by **lint**(1).

Certain conventional comments in the C source will change the behavior of **lint**(1):

/*NOTREACHED*/ At appropriate points stops comments about unreachable code. [This comment is typically placed just after calls to functions like **exit**(2)].

/*VARARGS*n*/ Suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first *n* arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/ Turns on the **-v** option for the next function.

/*LINTLIBRARY*/ At the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the **-v** and **-x** options.

lint(1) produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the **-c** option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

The behavior of the **-c** and the **-o** options allows for incremental use of **lint**(1) on a set of C source files. Generally, one invokes **lint**(1) once for each source file with the **-c** option. Each of these invocations produces a *.ln* file which corresponds to the *.c* file, and prints all messages that are about just that source file. After all the source files have been separately run through **lint**(1), it is invoked once more (without the **-c** option), listing all the *.ln* files with the needed **-l**x options. This will print all the inter-file inconsistencies. This scheme works well with **make**(1); it allows **make**(1) to be used to **lint**(1) only the source files that have been modified since the last time the set of source files were *lint*ed.

PARALLEL PROGRAMMING SUPPORT

If the environment has a variable *PARALLEL* with a value $N > 1$, **lint**(1) will process N input files in parallel. Using this feature can substantially reduce the time to **lint**(1) large programs having many input files.

FILES

LLIBDIR	The directory where the lint libraries specified by the **-l**x option must exist, usually */usr/lib*
LLIBDIR/lint[12]	First and second passes
LLIBDIR/llib-lc.ln	Declarations for C Library functions (binary format; source is in *LLIBDIR/llib-lc*)
LLIBDIR/llib-port.ln	Declarations for portable functions (binary format; source is in *LLIBDIR/llib-port*)
LLIBDIR/llib-lm.ln	Declarations for Math Library functions (binary format; source is in *LLIBDIR/llib-lm*)
TMPDIR/*lint*	Temporaries
TMPDIR	Usually */usr/tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

cc(1), cpp(1), make(1)

BUGS

exit(2), **setjmp**(3C), and other functions that do not return are not understood; this causes various lies.

NAME

list – produce C source listing from a common object file

SYNOPSIS

list [-V] [-h] [-F function] source-file . . . [object-file]

DESCRIPTION

The **list**(1) command produces a C source listing with line number information attached. If multiple C source files were used to create the object file, **list**(1) will accept multiple filenames. The object file is taken to be the last non-C source file argument. If no object file is specified, the default object file, *a.out*, will be used.

Line numbers will be printed for each line marked as breakpoint inserted by the compiler (generally, each executable C statement that begins a new line of source). Line numbering begins anew for each function. Line number 1 is always the line containing the left curly brace ({) that begins the function body. Line numbers will also be supplied for inner block redeclarations of local variables so that they can be distinguished by the symbolic debugger.

The following options are interpreted by **list**(1) and may be given in any order:

-V Print, on standard error, the version number of the **list**(1) command executing.

-h Suppress heading output.

-F *function* List only the named function. The **-F** option may be specified multiple times on the command line.

SEE ALSO

as(1), cc(1), ld(1)

DIAGNOSTICS

list(1) will produce the following error message if *name* cannot be read.

```
list: name: cannot open
```

If the source filenames do not end in *.c* , this message is produced:

```
list: name: invalid C source name
```

An invalid object file will cause this message to be produced:

```
list: name: bad magic
```

If some or all of the symbolic debugging information is missing, one of the following messages will be printed:

```
list: name: symbols have been stripped, cannot proceed
list: name: cannot read line numbers
list: name: not in symbol table.
```

The following messages are produced when **list**(1) has become confused by *#ifdef's* in the source file:

```
list: name: cannot find function in symbol table
list: name: out of sync: too many }
list: name: unexpected end-of-file
```

The following error message means that either symbol debugging information is missing or **list**(1) has been confused by C preprocessor statements.

```
list: name: missing or inappropriate line numbers
```

NOTES

Object files given to **list**(1) must have been compiled with the **-g** option of **cc**(1).

Because **list**(1) does not use the C preprocessor, it may be unable to recognize function definitions whose syntax has been distorted by the use of C preprocessor macro substitutions.

NAME

lorder – find ordering relation for an object library

SYNOPSIS

lorder file ...

DESCRIPTION

The input is one or more object or library archive *files* [see **ar**(1)]. The standard output is a list of pairs of object file or archive member names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by **tsort**(1) to find an ordering of a library suitable for one-pass access by **ld**(1). Note that the link editor **ld**(1) is capable of multiple passes over an archive in the portable archive format [see **ar**(4)] and does not require that **lorder**(1) be used when building an archive. The usage of the **lorder**(1) command may, however, allow for a slightly more efficient access of the archive during the link edit process.

The following example builds a new library from existing *.o* files.

```
ar –cr library `lorder *.o | tsort`
```

FILES

TMPDIR/*symref	Temporary files
TMPDIR/*symdef	Temporary files

TMPDIR is usually */usr/tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

ar(1), ld(1), tsort(1), ar(4)

WARNING

lorder will accept as input any object or archive file, regardless of its suffix, provided there is more than one input file. If there is but a single input file, its suffix must be *.o*.

NAME

nm – print name list of common object file

SYNOPSIS

nm [**–oxhvnefurpVT**] filename ...

DESCRIPTION

The **nm**(1) command displays the symbol table of each common object file, *filename*. *filename* may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Class Its storage class.

Type Its type and derived type. If the symbol is an instance of a structure or of a union, then the structure or union tag will be given following the type (e.g., struct-tag). If the symbol is an array, then the array dimensions will be given following the type (e.g., char[n][m]). Note that the object file must have been compiled with the **–g** option of the **cc**(1) command for this information to appear.

Size Its size in bytes, if available. Note that the object file must have been compiled with the **–g** option of the **cc**(1) command for this information to appear.

Line The source line number at which it is defined, if available. Note that the object file must have been compiled with the **–g** option of the **cc**(1) command for this information to appear.

Section For storage classes static and external, the object file section containing the symbol (e.g., text, data, or bss).

The output of **nm**(1) may be controlled by using the following options:

–o Print the value and size of a symbol in octal instead of decimal.

–x Print the value and size of a symbol in hexadecimal instead of decimal.

–h Do not display the output header data.

–v Sort external symbols by value before they are printed.

–n Sort external symbols by name before they are printed.

–e Print only external and static symbols.

–f Produce full output. Print symbols (.text, .data, .lib, and .bss), normally suppressed.

–u Print undefined symbols only.

–r Prepend the name of the object file or archive to each output line.

-p Produce easily parsable, terse output. Each symbol name is preceded by its value (blanks if undefined) and one of the letters **U** (undefined), **A** (absolute), **T** (text segment symbol), **D** (data segment symbol), **S** (user-defined segment symbol), **R** (register symbol), **F** (file symbol), or **C** (common symbol). If the symbol is local (non-external), the type letter is in lowercase.

-V Print the version of the **nm**(1) command executing on the standard error output.

-T By default, **nm**(1) prints the entire name of the symbols listed. Since object files can have symbol names with an arbitrary number of characters, a name that is longer than the width of the column set aside for names will overflow its column, forcing every column after the name to be misaligned. The **-T** option causes **nm**(1) to truncate every name which would otherwise overflow its column and place an asterisk as the last character in the displayed name to mark it as truncated.

Options may be used in any order, either singly or in combination, and may appear anywhere in the command line. Therefore, both **nm name -e -v** and **nm -ve name** print the static and external symbols in *name*, with external symbols sorted by value.

FILES

TMPDIR/* Temporary files

TMPDIR is usually */usr/tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam()** in **tmpnam**(3S)].

BUGS

When all the symbols are printed, they must be printed in the order they appear in the symbol table to preserve scoping information. Therefore, the **-v** and **-n** options should be used only in conjunction with the **-e** option.

SEE ALSO

as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4)

DIAGNOSTICS

"nm: name: cannot open"
If *name* cannot be read.

"nm: name: bad magic"
If *name* is not a common object file.

"nm: name: no symbols"
If the symbols have been stripped from *name*.

NAME

pdbx, dbx – parallel debugger

SYNOPSIS

pdbx [**–i**] [**–u**] [**–a**] [**–c** *file*] [**–I** *dir*] ... [*execfile* [*coredump*]]

pdbx [**–i**] [**–u**] [**–a**] [**–c** *file*] [[**–I** *dir*] ... **–O** *execfile*]
[**–Q** *execfile*] ... [**–C** *coredump*] ...

DESCRIPTION

Pdbx is a tool for source-level debugging and execution of both conventional and parallel programs. Currently, you can use Pdbx to source-level debug FORTRAN and C programs only, but the machine-level facilities of Pdbx can be used for any executable program. Pdbx is based on the UNIX 4.3bsd debugger, dbx, and performs almost identically to UNIX 4.3bsd dbx when used to debug conventional one-process, one-program applications. You can invoke Pdbx as either **pdbx** or **dbx** . Both names are links to the same program; however, forks and signals are handled appropriately for parallel debugging when Pdbx is invoked as **pdbx** . Refer to the *Pdbx User's Manual* for a list of differences between UNIX 4.3bsd dbx and Pdbx, and for differences dependent on invocation.

Use the first form of the **pdbx** command when debugging an application that consists of one or more processes running the same program. Use the second form when debugging applications that consist of multiple programs (e.g., client/server applications or processes that exec); each **–O** option specifies a program to debug.

execfile is an executable object file produced by the FORTRAN or C compiler when the **–g** option is specified on the compiler command line. It contains a symbol table that includes the names of all the source files translated by the compiler to create it. These files are available for perusal while using the debugger.

If a file named *core* exists in the current directory, or one or more *coredump* files are specified, you can use Pdbx to examine the state of the program when it faulted.

If a file named *.dbxinit* exists in the current directory, the debugger commands in *.dbxinit* are executed. Pdbx also checks for a *.dbxinit* in your home directory if one does not exist in the current directory.

When invoked, Pdbx processes command line options and *.dbxinit* information, and then prompts and waits for a command. Pdbx supports the following command line options:

–i Force the debugger to act as though standard input is a terminal (e.g., prompt for commands).

–u Preserve the case of identifiers for FORTRAN programs. The C compiler always preserves the case of identifiers. The FORTRAN compiler converts all identifiers to lowercase, unless the **–case** option is specified. If FORTRAN was invoked with the **–case** option, invoke Pdbx with the **–u** option to preserve case sensitivity.

–a	Run all processes asynchronously (i.e., in the background). As soon as Pdbx starts a process running, it prompts immediately for another command. No process may read from the terminal.
–c *file*	Execute the Pdbx commands in *file* before reading from standard input.
–I *dir*	Add *dir* to the list of directories that are searched when looking for a source file. Normally, Pdbx looks for source files in the current directory and in the directory where *execfile* is located. You can add a directory to the search path with the **use** command. When you specify more than one object file, each **–I** option applies to the program specified by the next **–O** option.
–O *execfile*	Identifies *execfile* as a program to debug. If there is only one program, invoke Pdbx without the **–O** option (use the first form of the **pdbx** command).
–Q *execfile*	Identifies *execfile* as a program not to debug. This is similar to the **–O** option, except that it specifies a program that has minimal debugging information. Use this option when the size of a program limits the amount of memory space available for debugging information.
–C *coredump*	Identifies *coredump* as a core file to analyze. Use the **–C** option when there is more than one program or more than one core file. The position of the **–C** option relative to the **–O** option on the command line is not important; Pdbx matches each core file with the appropriate object file.

Compiling Programs for Use with Pdbx

Specifying the **–g** option as an argument on the **fortran** (1) or **cc** (1) command line causes the compiler to produce symbolic information for Pdbx. If you explicitly invoke **ld** (1) to link your program, you must include the **–lg** option in the **ld** command line.

Code compiled with the **–g** option cannot be optimized using the compiler's **–O** option.

Debugging Multiprocess Applications

Pdbx can control multiple processes and their descendants (child processes). When a process is created (e.g., by means of the **run** command), it is assigned a process number. The first process created is process number 1, and the process created most recently by Pdbx is referred to as the current process. Certain Pdbx commands operate on the current process by default, but you can direct Pdbx to operate on a specified process or on all processes. You can list process numbers using Pdbx's **ps** command, and the current process designation can be changed using the **%** command.

To run a process in the background, append an ampersand (&) to the command that starts the process running (e.g., **run, cont, step**). Pdbx will prompt for another command without waiting for the process to stop.

Execution and Tracing Commands

run [*execfile*] [*args*] [< *filename*] [> *filename*] [&]
rerun [*execfile*] [*args*] [< *filename*] [> *filename*] [&]
Start executing *execfile*, passing *args* as command line arguments. You can use redirection in the usual manner. When you enter the **rerun** command without arguments, the previous argument list is passed to the program; otherwise it is identical to **run**. **run** and **rerun** terminate any existing processes before creating the process to execute.

NOTE: You must specify the *execfile* parameter when more than one *execfile* was specified when Pdbx was invoked; otherwise it must be omitted.

release *%procnum*
Release the specified process from Pdbx control and continue execution without further debugger control. Child processes whose setuid bit is set can be released immediately after being stopped by a fork to allow debugging of the parent to continue.

create [*execfile*] [*args*] [< *filename*] [> *filename*]
Create a process as with **run**, but do not run the process and do not terminate any existing processes. The created process can be run using the **cont** command.

terminate [*%n*]
terminate all
Terminate the specified process (defaults to the current process). The **ps** command does not list terminated processes.

ps [*%n*]
Report the status, process id, and parent process of process *n*. If no process is specified, print a list of all processes, indicating the status of each. For core dumps, **ps** reports which type of signal caused the program to abort.

%n Change the current process to process *n* .

trace [*procid*] [**in** *procedure*] [**if** *condition*]
trace [*procid*] *source-line-number* [**if** *condition*]
trace [*procid*] *procedure* [**in** *procedure*] [**if** *condition*]
trace [*procid*] *expression* **at** *source-line-number* [**if** *condition*]
trace [*procid*] *variable* [**in** *procedure*] [**if** *condition*]
Print trace information during program execution. Use the **delete** command to turn tracing off.

For multiprocess applications, the optional *procid* parameter specifies which process to trace (defaults to all processes associated with the current source file). A *procid* of the form *%n* directs Pdbx to trace process *n* . A *procid* of the form *%execfile* directs Pdbx to trace all processes created from *execfile* .

The next argument describes what to trace. If it is a *source-line-number*, the line is printed just before execution. Source line numbers in a file other than the current one must be preceded by the name of the file in quotes and a colon (e.g., "mumble.c":17).

If the argument is a procedure or function name, every time it is called, information is printed telling what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted, and if it's a function, the value it is returning is also printed.

If the argument is an expression with an **at** clause, the value of the expression is printed just before the identified source line is executed.

If the argument is a variable, the name and value of the variable is printed whenever it changes. Execution is substantially slower during this form of tracing.

If no argument is specified, all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause "**in** *procedure*" restricts tracing information to be printed only while executing inside the given procedure or function.

condition is a Boolean expression and is evaluated prior to printing the tracing information; if it is false, the information is not printed.

stop [*procid*] **if** *condition*
stop [*procid*] **at** *source-line-number* [**if** *condition*]
stop [*procid*] **in** *procedure* [**if** *condition*]
stop [*procid*] *variable* [**if** *condition*]
stop all

Stop execution when the given line is reached, a procedure or function is called, a variable is changed, or the condition is true. **stop all** immediately stops all running processes; all other forms of the **stop** command set a breakpoint.

For multiprocess applications, the optional *procid* parameter specifies which process the breakpoint applies to. By default, the breakpoint is set in all processes created from the current program, and only the process that encounters the breakpoint is stopped. A *procid* of the form *%n* specifies that the breakpoint applies only to process *n*. If the *procid* is **all**, all processes are stopped if any process encounters the breakpoint. A *procid* of the form *%execfile* specifies that the breakpoint applies to each process created from *execfile* .

If a process is running, the **stop** command does not take effect in that process until it is stopped.

status [> *filename*]

Print the currently active tracepoints and breakpoints.

delete *cmdnumber* ...

Cancel the tracepoints or breakpoints associated with *cmdnumber* . Use the **status** command to print the current tracepoints and breakpoints, and their associated command numbers.

catch [*signal*]
ignore [*signal*]

Start or stop trapping a signal before it is sent to the program. This is useful when a program being debugged handles signals such as interrupts. A signal may be specified by number or by a name (e.g., SIGINT). Signal names are case insensitive and the "SIG" prefix is optional. By default all signals are trapped except SIGCONT, SIGCHLD, SIGALRM, SIGHUP, and SIGKILL. When entered without parameters, **catch** lists the signals and events that will be caught, and **ignore** lists the signals and events that will be ignored.

catch *event*
ignore *event*

Catch or ignore the specified event, which may be **fork**, **exec**, or **exit**. By default, execs and exits are caught and forks are ignored. If you direct Pdbx to catch forks, the child process is stopped immediately after a fork, but the parent is not stopped. If you direct Pdbx to catch execs, the process is stopped just before the new program begins execution. If you direct Pdbx to catch exits, the process is stopped just before it is removed by the system; no matter how you resume execution of the process, it will exit.

signal [*procid*] *signal*

Send the specified signal to the specified process (defaults to the current process). *procid* may be **all** or *%n*.

cont [*procid*] [*signal*] [**to** *source-line-number*] [&]

Continue execution from where it stopped. For multiprocess applications, the optional *procid* parameter specifies which process to continue (defaults to the current process). *procid* may be **all** or *%n*.

If a signal is specified, the process continues as though it received the signal (any signal previously caught is canceled). A signal of zero indicates that execution should continue as if the process had never been stopped. Otherwise, the process is continued with whatever signal was just received. If the process has multiple signals pending, the *signal* parameter overrides only the signal that was caught.

If the optional *source-line-number* parameter is specified, execution continues until the specified source line is executed, then stops.

step [*procid*] [&]

Execute one source line. Any signal previously caught is canceled. For multiprocess applications, the optional *procid* parameter specifies which process to execute. *procid* may be **all** or *%n*.

next [*procid*] [&]

Execute up to the next source line. Any signal previously caught is canceled. *procid* may be **all** or *%n*. The difference between **next** and **step** is that if the line contains a call to a procedure (or function), the **step** command will stop at the beginning of that procedure, while the **next** command will stop after completion of the procedure.

return [*procedure*]
Continue until a return to *procedure* is executed, or until the current procedure returns if none is specified.

call *procedure()*
call *procedure(parameters)*
Execute the object code associated with the named procedure or function.

Printing Variables and Expressions

Names are resolved first using the static scope of the current function, then using the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message `[using` *qualified name*`]` is printed. This may be overridden by qualifying an identifier with a block name, e.g., "*module.variable*". A *module* consists of the subroutines and data inside a source file, and is named by the file name without the *.c* or *.f* suffix.

The C compiler always preserves the case of identifiers. The FORTRAN compiler converts all identifiers to lowercase (unless the FORTRAN command line option **–case** is specified). Specifying **–u** when invoking Pdbx preserves the case of identifiers for FORTRAN programs.

Expressions are specified with a C-like syntax. Indirection can be denoted using either a prefix "*" or a postfix "^", and array expressions are subscripted by brackets ([ ]). The field reference operator (".") can be used with pointers as well as records, so a period can be used in place of the C operator "->" (but "->" is also supported).

For multiprocess applications, an expression of the form *n*:*expr* yields the value of *expr* when interpreted in the context of process *n* . Process *n* must be stopped when the expression is evaluated. If the *n*: prefix is omitted, the current process is used.

Types of expressions are checked; the type of an expression may be overridden by using the syntax (*expression*)*type-name*. When there is no corresponding named type, the special constructs &*type-name* and $$*tag-name* can be used to represent a pointer to a named type or C structure tag.

assign *variable* = *expression*
Assign the value of the expression to the variable.

dump [*procedure*] [> *filename*]
Print the names and values of variables in the given procedure, or the current one if none is specified. If *procedure* is ".", Pdbx dumps all active variables.

print *expression* [, *expression*] ...
Print the values of the specified expressions.

print fpustack
Display control and data register information for the 80387 floating-point coprocessor.

func [*procedure*]
Change the current function. If none is specified, print the current function. Changing the current function with **func** implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

up [*count*]
down [*count*]
Move the current function up or down the stack *count* levels. The default *count* is 1. **up** moves toward the main program; **down** moves toward the most recently called procedure. The use of **up** and **down** does not change the current source file.

whatis *name*
Print the declaration of the given name, which may be qualified with block names as above.

which *identifier*
Print the full qualification of the given identifier, i.e., the outer blocks that the identifier is associated with.

where [*%n*]
Print a list of the active procedures for the specified process (defaults to the current process).

whereis *identifier*
Print the full qualification of all the symbols in the current program whose names match the given identifier. The order in which the symbols are printed is not meaningful.

Accessing Source Files

/regular expression[*/*]
?regular expression[*?*]
Search forward or backward in the current source file for the given pattern.

edit [*filename*]
edit *procedure-name*
Invoke an editor on *filename* or the current source file if none is specified. If you specify the name of a procedure or function, the editor is invoked on the file that contains it. The environment variable **EDITOR** should be set to the name of the desired editor — e.g., */usr/bin/vi* .

file [[*%execfile*] *sourcefile*]
Change the current source file name to *sourcefile*. If none is specified, the current source file name is printed. If an object file is specified, Pdbx searches for *sourcefile* in the source directories listed for the specified program.

list [*source-line-number* [, *source-line-number*]]
list *procedure*
List the lines in the current source file from the first line number to the second inclusive. If no lines are specified, the next 10 lines are listed. If the name of a procedure or function is given, Pdbx lists a

10-line window centered around the first statement of the specified procedure or function. The default number of lines to list is specified by the debugger variable *$listwindow*, which can be set using the **set** command.

use [*%execfile*] [*directory* ...]
Set or display the list of directories to be searched when looking for source files. If an object file is specified, the **use** command applies to the indicated program.

Command Aliases and Variables

alias *name cmdname*
alias *name "cmdstring"*
alias *name* (*parameter* [, *parameter*] ...) *"cmdstring"*
alias [*name*]
Define an alias for the specified command name or command string. If no command name or string is specified, display the definition of the specified alias, or of all aliases if none is specified.

When commands are processed, Pdbx first checks to see if the word is an alias for either a command or a string. If it is an alias, Pdbx treats the input as though the corresponding string (with values substituted for any parameters) had been entered. For example, use the following command to define an alias **rr** for the command **rerun** :

```
alias rr rerun
```

To define an alias called **b** that sets a stop at a particular line enter the following:

```
alias b(x) "stop at x"
```

Subsequently, the command **b(12)** will expand to **stop at 12.**

set [*name* [= *expression*]]
The **set** command defines or displays values for debugger variables. The names of these variables cannot conflict with names in the program being debugged, and are expanded to the corresponding expression within other commands. The following variables have special meanings:

$fpasingle
When set, Pdbx displays values in floating-point registers as single-precision numbers.
$fpustackall
When set, Pdbx displays nonzero floating-point stack entries even when in empty state (80387 FPU only).

$hexchars
$hexints
$hexoffsets
$hexstrings
When set, Pdbx prints out characters, integers, offsets from registers, or character pointers, respectively, in hexadecimal.

$hexin
$octin
When set, Pdbx expects integers to be entered in hexadecimal or octal, respectively. If $hexin is set, $octin is ignored.

$listwindow
The value of this variable specifies the number of lines to list around a function or the number of lines listed when the **list** command is given without any parameters. Its default value is 10.

$noframe
When set, Pdbx does not attempt to follow the chain of call frames on the stack. This allows access to global variables and register contents in code that does not establish conventional stack frames.

$nostrict
When set, Pdbx relaxes type checking rules for **call** and **assign** commands.

$whichreg
When set, Pdbx includes the register name with the information displayed by the **whatis** command for the specified variable. Your program must declare the storage class for the variable as type `register`. Valid for C programs only.

unalias *name*
Remove the alias with the given name.

unset *name*
Delete the debugger variable associated with *name.*

Machine Level Commands

tracei [*procid*] [*address*] [**if** *cond*]
tracei [*procid*] [*variable*] [**at** *address*] [**if** *cond*]
Turn on tracing using a machine instruction address.

stopi [*procid*] [**at**] *address* [**if** *cond*]
Stop on accesses to the specified address. **stopi at** *address* sets a breakpoint at the machine instruction at the specified address. **stopi** *address* traces changes to the variable at the specified address.

stepi [*procid*] [&]
nexti [*procid*] [&]
Single-step as in **step** or **next**, but do a single instruction rather than a source line.

listi [*source-line-number* [, *source-line-number*]]
listi *procedure*
Same as **list**, but print the machine instructions that correspond to the specified source lines.

address , *address*/ [*format*]
address / [*count*] [*format*]
Print the contents of memory starting at the first *address* and continuing up to the second *address* or until *count* items are printed. If the address is a period (.), Pdbx uses the address that follows the one most recently printed. The *format* specifies how values are displayed; if it is omitted, the last specified format is used. The initial value of *format* is **X** . Pdbx supports the following formats:

i	print the machine instruction
d	print a short word in decimal
D	print a long word in decimal
o	print a short word in octal
O	print a long word in octal
x	print a short word in hexadecimal
X	print a long word in hexadecimal
b	print a byte in octal
c	print a byte as a character
s	print a string of characters terminated by a null byte
f	print a single precision real number
g	print a double precision real number

address = [*format*]
Print the value of the specified *address* in the specified format (refer to previous list of supported formats). For example, entering the command **0xffff=D** prints the decimal equivalent of 0xffff.

Symbolic addresses are specified by preceding the name with an ampersand (&). Register names must be preceded by a dollar sign ($). The register names are: eax, edx, ecx, ebx, ebp, esi, edi, esp, eip, and eflags. The floating-point registers for the 80387 are st0–st7. The FPA (Weitek 1167) registers are fp1–fp31. In addition, the aliases $pc, $fp, and $sp refer to the program counter, frame pointer, and stack pointer, respectively. Addresses may be expressions made up of other addresses and the operators "+", "-", and indirection (unary "*").

Miscellaneous Commands

help Print synopsis of frequently used Pdbx commands.

quit Terminate all processes and exit Pdbx.

sh [*command-line*]
Create a new shell and pass it the specified command line shell for execution, or execute a new shell if no command line is specified. The value of the **SHELL** environment variable determines which shell is used.

suspend
Suspend Pdbx and return control to the shell that invoked it (this command is not supported by the Bourne shell). The effect of entering this command is analogous to the effect of entering CTRL-Z in a shell. Any running process continues to run until it hits a breakpoint or receives a signal.

source *filename*
Read Pdbx commands from the specified *filename*.

FILES

a.out	object file
.dbxinit	initialization commands
core	core dump file

SEE ALSO

cc(1), fortran(1)
Pdbx User's Manual

COMMENTS

Pdbx suffers from the same "multiple include" malady as **sdb.** If you have a program consisting of a number of object files and each is built from source files that include header files, the symbolic information for the header files is replicated in each object file. Since approximately one debugger start-up is done for each link, having the linker (**ld**) reorganize the symbol information would not save much time, but it would reduce some of the disk space used.

This problem is an artifact of the unrestricted semantics of #include's in C. For example, an include file can contain static declarations that are separate entities for each file in which they are included.

NAME

size – print section sizes in bytes of common object files

SYNOPSIS

size [**–n**] [**–f**] [**–o**] [**–x**] [**–V**] files

DESCRIPTION

The **size**(1) command produces section size information in bytes for each loaded section in the common object files. The size of the text, data, and bss (uninitialized data) sections is printed, as well as the sum of the sizes of these sections. If an archive file is input to the **size**(1) command, the information for all archive members is displayed.

The **–n** option includes NOLOAD sections in the size.

The **–f** option produces full output, that is, it prints the size of every loaded section, followed by the section name in parentheses.

Numbers will be printed in decimal unless either the **–o** or the **–x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **–V** flag will supply the version information on the **size**(1) command.

SEE ALSO

as(1), cc(1), ld(1), a.out(4), ar(4)

DIAGNOSTICS

size: name: cannot open
: If *name* cannot be read.

size: name: bad magic
: If *name* is not an appropriate common object file.

WARNING

Because the size of bss sections is not known until link-edit time, the **size**(1) command will not give the true total.

NAME

strip – strip symbol and line number information from a common object file

SYNOPSIS

strip [–l] [–x] [–b] [–r] [–V] filename ...

DESCRIPTION

The **strip**(1) command strips the symbol table and line number information from common object files, including archives. Once this has been done, no symbolic debugging access will be available for that file. Therefore, this command is normally run only on production modules that have been debugged and tested.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

–l Strip line number information only; do not strip any symbol table information.

–x Do not strip static or external symbol information.

–b Same as the **–x** option, but also do not strip scoping information (e.g., beginning and end of block delimiters).

–r Do not strip static or external symbol information or relocation information.

–V Print the version of the **strip**(1) command executing on the standard error output.

If there are any relocation entries in the object file and any symbol table information is to be stripped, **strip**(1) will complain and terminate without stripping *filename* unless the **–r** option is used.

If the **strip**(1) command is executed on a common archive file [see **ar**(4)], the archive symbol table will be removed. The archive symbol table must be restored by executing the **ar**(1) command with the **s** option before the archive can be link-edited by the **ld**(1) command. **strip**(1) will produce appropriate warning messages when this situation arises.

strip(1) is used to reduce the file storage overhead taken by the object file.

FILES

TMPDIR/strp* Temporary files

TMPDIR is usually */usr/tmp* but can be redefined by setting the environment variable **TMPDIR** [see **tempnam**() in **tmpnam**(3S)].

SEE ALSO

ar(1), as(1), cc(1), ld(1), tmpnam(3S), a.out(4), ar(4)

DIAGNOSTICS

strip: name: cannot open
If *name* cannot be read.

strip: name: bad magic
If *name* is not an appropriate common object file.

strip: name: relocation entries present; cannot strip
If *name* contains relocation entries and the **–r** flag is not used, the symbol table information cannot be stripped.